Supplementary: Shading Atlas Streaming

A PARALLEL MEMORY MANAGEMENT

Our memory management strategy uses three distinct phases, a request phase, a provision phase and an assignment phase. The phases have the following responsibilities:

- The *request phase* frees unused blocks and counts how many blocks will be allocated.
- The *provision phase* allocates superblocks depending on how many blocks are already available and have to be newly allocated from superblocks.
- The *assignment phase* uses the information prepared in the previous two phases to allocate the blocks.

We manage free memory in resource stacks. The subdivision into phases, where either allocation or deallocation occur, lets us operate each resource stack by atomically increasing or decreasing stack counters, leading to a lock-free algorithm that is friendly to SIMD operation. We use one stack for each block size and an additional stack for unused superblocks. Free memory is either available on a block stack, at or inside an unused superblock. In the former case, the memory can only be used for a block of the corresponding size, while memory inside an unused superblock still has the potential to be allocated for any block size.

Superblocks are not freed separately, possibly leading to memory fragmentation. As soon as blocks have been allocated from within a superblock, they are managed on the resource stacks and cannot change their size. These blocks will only be reused if the same block size is requested again. An efficient parallel algorithm that finds free blocks that can be merged into larger ones is hard to implement. Instead, we capitalize on a property of MPEG encoding that allows us to break with our requirement for temporal coherence. The problem is solved with a garbage collection step that frees all memory and newly allocates it. Doing this in correspondence with MPEG I-frames leads to ideal use of the video encoding bandwidth, since the temporal coherence is still ensured between I-frames.

A.1 Request phase

The request phase runs in parallel for all patches as outlined in algorithm 1. Patches that have become invisible are inserted into a block resource stack corresponding to their level (lines 3 to 5). Patches that have become visible increase an atomic request counter corresponding to their level (line 7) storing the resulting *slot number P.s*. Patches can also be reallocated, i.e., removed and re-inserted in the same frame, where both conditions evaluate true (lines 2 and 6). At the end of the phase, we have determined the total number of blocks required to be allocated for each level, and each patch stores its slot, i.e., the value it has drawn from the request counter.

A.2 Assignment phase

We first investigate the assignment phase to see which information the provision phase has to provide. The assignment phase runs in parallel for all patches as outlined in algorithm 2. Based on its slot, each patch determines whether its request is served by a list of superblocks (lines 4 to 6) or the block resource stack (lines 8 to 13).

ALGORITHM 1: Request Phase

```
/* the request phase runs in parallel over all patches
   that are visible or allocated                              */
1  for all patches P that are visible or allocated do
       /* the patch should be deallocated as it became
          invisible or is reallocated                          */
2      if P.toDeallocate then
           /* the block b assigned to the patch P is pushed
              onto the block stack for level l                 */
3          $p \leftarrow$ atomicAdd(blockStack[l].count, 1)
4          blockStack[l].entries[$p$] $\leftarrow$ $P$.b
5          $P$.b $\leftarrow$ INVALID
       /* the patch should be allocated as it became visible
          or is reallocated                                    */
6      if P.toAllocate then
           /* a slot s is reserved as the number of blocks to
              be allocated for level l is counted              */
7          $P.s \leftarrow$ atomicAdd(blocksRequested[l], 1)
```

ALGORITHM 2: Assignment Phase

```
/* the assignment phase runs in parallel over all visible
   patches                                                    */
1  for all visible patches P do
       /* if the patch is allocating a block                   */
2      if P.toAllocate then
           /* check if the slot falls within range allocating
              from the superblock list                         */
3          if $P.s <$ slotCount[l] then
               /* we compute the position in the list of
                  allocated superblocks                        */
4              $s \leftarrow P.s +$ slotOffset[l]
5              $i \leftarrow \left\lfloor \frac{s}{\mathrm{BpSB(l)}} \right\rfloor$
               /* get the corresponding superblock and compute
                  the block id within the superblock           */
6              $P$.b $\leftarrow$ [sb[$\mathrm{l}_x$][$i$], $s - i \cdot$ BpSB(l)]
7          else
               /* we try to allocate from the stack            */
8              $p \leftarrow$ atomicAdd(blockStack[l].count, -1) - 1
9              if ($p < 0$) then
                   /* if that failed, we are out of memory     */
10                 atomicAdd(blockStack[l].count, 1)
11                 $P$.b $\leftarrow$ INVALID
12             else
13                 $P$.b $\leftarrow$ blockStack[l].entries[$p$]
```

This strategy is not only lock-free, it also deals with provisioning on a superblock level, avoiding tedious bookkeeping during bulk allocations. Recycling happens on the level of individual blocks, but uses efficient atomically operated stacks to manage the recycled blocks.

To determine which of the two methods is used for a given patch, the *slotCount* for the patch's level **l** is compared to the patch's slot number *P.s* in line 5. If the patch is allocated from a superblock, the patch uses its slot and the *slotOffset* to compute which superblock is provisioned to fulfill its request (lines 4 and 5). The *BpSB*(**l**) function gives the number of blocks a superblock can store for level **l** if all blocks were the same level. The slot can directly be used to compute the patch's offset in the superblock (line 6). By reading the superblock index corresponding to the slot from the *sb* array, the allocation is fulfilled (also line 6). The slot number *P.s* of the patch was determined in the request phase, but the remaining information about allocation from superblocks has to come from the provision phase. This information includes the *slotCount*, *slotOffset* and *sb* arrays, where the later is shared for all levels with the same block width.

If the slot index *P.s* points to an entry outside the bounds of the array, it draws a block from the resource stack (starting at line 8). The atomic decrement (line 8) returns the position *p* on the stack of the block that is allocated. If the position *p* is negative, allocation fails, since there are no blocks left on the stack (line 9). In this case we revert the atomic decrement (line 10) and remember that allocation failed (line 11). Otherwise, we simply get the block from the resource stack (line 13).

A.3 Provision phase

The provision phase determines how many requests from patches can be served from the block resource stack. For the remaining patches, it draws free superblocks from the superblock resource stack. The indices of the provisioned superblocks are stored in the sb array, allowing patches to pick their respective slots from this array in the assignment phase. In most cases, the last superblock drawn from the resource stack will not be fully filled by the requests. Therefore, we memorize the fill-rate of the last superblock for each level, and top it up in the next frames before requesting a new superblock for that level.

The provision phase operates in parallel per block width as outlined in algorithm 3. It allocates superblocks for all levels with the same block width in the *sb* array as shown in figure 1. The superblocks for one specific width are subdivided into columns of this width. Within the columns blocks are allocated in order with decreasing size. This ensures that there will be no alignment problems, such as a block split in half as it should begin at the end of one column and end at the beginning of the next column.

First, the number of superblocks to be allocated has to be computed. We do this by add up the number of pixel lines *r* within the columns (lines 2 to 4) This can also be thought of as the number of blocks of the smallest block size - with one pixel height. Since we carry over the last superblock from the last frame that has not been completely filled, we first consider the remaining space within this superblock (line 2). The *carryOver offset* stores how many square blocks of this size have already been allocated in the block. Using the number of blocks that would fit in the superblock (*BpSB*) we can compute the remaining space. At this point *r* is zero or negative

ALGORITHM 3: Provision Phase

```
/* the provision phase runs in parallel over all block
   widths                                                */
1  for all block widths $\mathbf{l}_x$ do
       /* compute how many superblocks need to be newly
          allocated                                      */
2      $r \leftarrow \left(\text{carryOver}[\mathbf{l}_x].\text{offset} - \text{BpSB}\left([\mathbf{l}_x\ \mathbf{l}_x]^T\right)\right) \cdot 2^{\mathbf{l}_x}$
3      for all block heights $\mathbf{l}_y$ do
4          $r$ += max(0, blocksRequested[$\mathbf{l}$] - blockStack[$\mathbf{l}$].count) $\cdot 2^{\mathbf{l}_y}$
       /* try to allocate as many superblocks as required and
          fill in the sb array                           */
5      sb[$\mathbf{l}_x$][0] $\leftarrow$ carryOver[$\mathbf{l}_x$].id
6      superblockCount $\leftarrow$ 1
7      for $i \leftarrow 1$ to $\left\lceil \frac{r}{BpSB\left([\mathbf{l}_x\ 0]^T\right)} \right\rceil$ do
8          $p \leftarrow$ atomicAdd(superblockStack.count, -1) - 1
9          if $p < 0$ then
10             atomicAdd(superblockStack.count, 1)
11             break
12         sb[$\mathbf{l}_x$][$i$] $\leftarrow$ superblockStack.entries[$p$]
13         superblockCount + = 1
       /* compute per level information of the slot range
          within the sb array and store results in the
          slotOffset and slotCount arrays                */
14     $o \leftarrow$ carryOver[$\mathbf{l}_x$].offset
15     $a \leftarrow$ superblockCount $\cdot$BpSB $\left([\mathbf{l}_x\ \mathbf{l}_x]^T\right) - o$
16     for $\mathbf{l}_y \leftarrow \mathbf{l}_x$ to $1$ do
17         $r \leftarrow$ clamp(blocksRequested[$\mathbf{l}$] - blockStack[$\mathbf{l}$].count, 0, $a$)
18         slotOffset[$\mathbf{l}$] $\leftarrow o$
19         slotCount[$\mathbf{l}$] $\leftarrow r$
20         blocksRequested[$\mathbf{l}$] $\leftarrow$ 0
21         $o \leftarrow 2 \cdot (o + r)$
22         $a \leftarrow 2 \cdot (a - r)$
       /* put remaining rectangular blocks on their
          respective stacks                              */
23     for $\mathbf{l}_y \leftarrow 1$ to $\mathbf{l}_x - 1$ do
24         $o \leftarrow \lceil \frac{o}{2} \rceil$
25         if $o$ is odd then
26             $p \leftarrow$ atomicAdd(blockStack[$\mathbf{l}$].count, 1)
27             blockStack[$\mathbf{l}$].entries[$p$] $\leftarrow$ [$o$%BpSB ($\mathbf{l}$),
                 sb[$\mathbf{l}_x$][superblockCount - 1]]
       /* store carryOver information for the next frame */
28     carryOver[$\mathbf{l}_x$].offset = $o$
29     carryOver[$\mathbf{l}_x$].id = sb[$\mathbf{l}_x$][superblockCount - 1]]
```

as this space has already been allocated and will be used up first. Before the first frame the carryOver offset has to be initialized with the value from *BpSB* in order to initialize *r* correctly in the first frame. In the following loop, we then add up the number of pixel lines necessary for each block size, counting only requested blocks, that cannot be allocated from the corresponding block stack (lines 3 and 4).

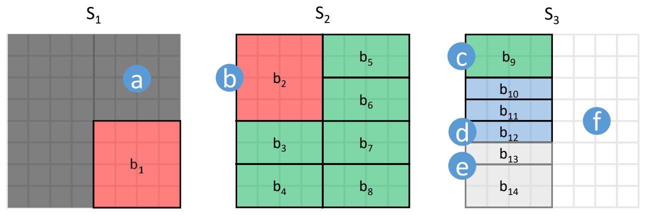

Fig. 1. Example of the provisioning phase of the parallel memory allocation and the corresponding lines in algorithm 3. (a) The superblock carried over from the last frame S_1 is partially filled (line 2). The actual allocation of superblocks requested for each block size (b to d) is done in lines 3 to 22, separated into three distinct steps: counting how many superblocks are required (lines 3 and 4), allocating the superblocks (lines 5 to 13) and then assigning the superblocks to each level (lines 14 to 22). (b) The allocation of the 4×4 blocks in the first and second superblock S_2 is done in the first iteration of the loops in lines 3 to 4 and 16 to 22). (c) For the 4×2 blocks, another superblock S_3 is provisioned in the second iteration in the loops in lines 3 to 4 and 16 to 22. (d) The 4×1 blocks fit into the remaining space of S_3 (third iteration in the loops in lines 3 to 4 and 16 to 22). (e) Remaining rectangular blocks are put on block stacks (lines 23 to 27). (f) The remaining space in S_3 will be used in the provisioning phase of the next frame (lines 28 to 29).

Next, we allocate the required superblocks. The *sb* array is started with the carried over superblock (lines 5 and 6). The following loop (lines 7 to 13) allocates the new superblocks from the superblock stack in the same way allocation works for block stacks in the assignment phase (lines 8 to 12). If we run out of memory, the loop is aborted early (line 11), resulting in less superblocks allocated (*superblockCount*) than requested.

The next step is to fill in the slotOffset and slotCount arrays for the assignment phase. We use an offset variable *o* that stores the offset and an available block count variable *a* that stores the number of still available blocks within the superblocks. The variables are initialized with values for the largest (square) block size (lines 14 and 15). In the following loop (lines 16 to 22) the offsets are assigned in the discussed order from largest to smallest block size. The number of blocks that can be allocated from the superblocks needs to be clamped between zero and the number of available blocks (line 17). This value and the offset are then stored in the *slotCount* and *slotOffset* arrays and the number of requested blocks *blocksRequested* is reset for the next frame (lines 18 to 20). For the next iteration the offset *o* and available blocks *a* are updated and doubled as the next iteration's block size is halved (lines 21 and 22).

With the arrays *sa*, *slotCount* and *slotOffset* prepared for the assignment phase, the remaining task is to prepare the last superblock for the next frame. The alignment needs to be fixed to start with the allocation of square blocks in the next frame. In a loop we deallocate any non-square blocks that remain at the end of the last superblock (lines 23 to 27). This is done by looping over the block sizes in the opposite direction - from smallest to largest - and halve the final offset *o* again (lines 23 and 24) until we arrive back at the square block size. If the offset *o* cannot be halved without a remainder (line 25), we deallocate a block of the current size (lines 26 and 27) in the same way it is done in the request phase. The modulo operator % is used to calculate the block's index within the superblock. Finally, we store the resulting final offset and the last superblock's id for the next frame (lines 28 and 29).