Volume xx (200y), Number z, pp. 1–11

InK-Compact: In-Kernel Stream Compaction and Its Application to Multi-Kernel Data Visualization on General-Purpose GPUs

D. M. Hughes[1] I. S. Lim[1] M. W. Jones[2] A. Knoll[3] and B. Spencer[2]

[1]Bangor University, Bangor, UK
[2]Swansea University, Swansea, UK
[3]University of Texas at Austin, Texas, USA

Abstract

Stream compaction is an important parallel computing primitive that produces a reduced (compacted) output stream consisting of only valid elements from an input stream containing both invalid and valid elements. Computing on this compacted stream rather than the mixed input stream leads to improvements in performance, load balancing, and memory footprint. Stream compaction has numerous applications in a wide range of domains: e.g., deferred shading, isosurface extraction, and surface voxelization in computer graphics and visualization. We present a novel In-Kernel stream compaction method, where compaction is completed before leaving an operating kernel. This contrasts with conventional parallel compaction methods that require leaving the kernel and running a prefix sum kernel followed by a scatter kernel. We apply our compaction methods to ray-tracing-based visualization of volumetric data. We demonstrate that the proposed In-Kernel Compaction outperforms the standard out-of-kernel Thrust parallel-scan method for performing stream compaction in this real-world application. For the data visualization, we also propose a novel multi-kernel ray-tracing pipeline for increased thread coherency and show that it outperforms a conventional single-kernel approach.

Categories and Subject Descriptors (according to ACM CCS):

1. Introduction

In scientific visualization and other computational fields, a major challenge is data reduction, the process of identifying and processing interesting subsets of a larger input set. Parallel computing relies heavily on certain computational paradigms for numerous applications, including prefix-sum [HSO07, SHZO07], parallel sorting [SHG09] and compaction [HSO07, HS86].

Parallel algorithms typically generate data containing both wanted and unwanted elements for further processing steps. It is important to compact the data prior to follow-on processing in order to reduce the computational working set and improve load balancing. Stream compaction generates a smaller output stream containing only wanted elements from the input data stream consisting of the mixed elements. This effectively performs parallel data reduction, extracting and processing the desired subset of data (Fig. 1).

Stream compaction has a wide range of applications in parallel computing and GPGPU (General-Purpose Graphical Processing Units) computing. In scientific data visualization, for instance, extracting level sets or isosurfaces from volume data is a classic data reduction application. A stream compaction algorithm can identify a working set of voxels from a domain decomposition structure, compact the output, and pass that to a separate kernel. Applications of this include data-parallel mesh extraction, computation of surface area or volume, data compression, and adaptive volume rendering. In rendering, deferred shading is another example; one can use stream compaction to obtain the subset of pixels whose rays intersect geometry. When we send a compacted stream to the shader, we know exactly which rays require shading, and that each thread will perform similar action on the working set. Without compaction, many threads sit idle, resulting in poor performance.

This work is motivated by the need for multiple kernels in *optimized* GPGPU computing in general and rendering for visualization in particular. Although it is possible to pack an entire rendering pipeline into a single kernel [HL09], this approach suffers from threads being in different states due to large code branches.

By dividing up a kernel into smaller kernels, multi-kernel pipelines allow for better concurrent execution on the GPU and ensure that each kernel is light-weight and modular. To benefit from these, the output of each kernel needs to be compacted. However, the conventional compaction methods must prepare a preliminary output-array, perform a N-wide prefix sum and then perform a scatter operation [HSO07]. To avoid these extra burdens, we advocate an in-kernel approach that completes compaction before leaving an operating kernel (see Fig. 1). We demonstrate the effectiveness of our in-kernel compaction with a multi-kernel pipeline for real-time isosurface ray-casting of volumetric data, which benefits from large reductions in both volume and image domains and employs multiple successive kernels performing stream compaction.

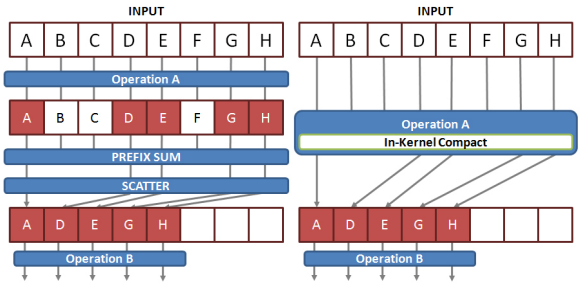
The paper is organised as follows. In section 2 we detail our In-Kernel Compaction method (InK-Compact). In section 3 we outline our multi-kernel rendering pipeline, reliant on fast compaction, comprised of a BVH (Bounding Volume Hierarchy) purpose-built per-frame for the current isovalue, ray generation, ray traversal of the BVH, ray-isosurface intersection and finally shading of hits to the screen. In section 4 we report the results of our rendering pipeline using the proposed InK-Compact and compare it against state-of-art compaction methods. Overall, we find our InK-Compact approach yields better performance than popular out-of-kernel libraries, such as Thrust [HB10] and Chag:PP [BOA09].

1.1. Background

Stream compaction takes an input stream $\mathbf{X}_{[0...N-1]}$ with a predicate $\mathbf{P}_{[0...N-1]}$ and generates an output stream $\mathbf{Y}_{[0...M-1]}$ that only contains the input elements $\mathbf{X}_{[i]}$ where $\mathbf{P}_{[i]}$ is true; the output stream preserves the ordering of the input elements and $M \leq N$.

Implementations of stream compaction usually consist of two operations, a *scan* and a *scatter* [BOA09, HSO07, HB10, SHZO07]. A scan is a (parallel) exclusive prefix sum that performs on a temporary stream containing '1' for each valid element in the input $\mathbf{X}_{[0...N-1]}$ and '0' for each invalid element. This results in a stream containing, for each element, the total number of valid elements preceding it, which is used as the destination address by a scatter to copy each valid element into the output $\mathbf{Y}_{[0...M-1]}$. Scattering is the process of reading (gathering) the input and new output locations then performing random parallel writes to the output buffer.

While this operation appears simple, in a parallel system it is a non-trivial problem, to which scan-scatter approach has been widely employed [HSO07, SHZO07]. The Thrust [HB10] library is a collection of implementations for various parallel primitive methods, including scan-scatter compaction. Given its popularity, we compare our compaction method against Thrust. Unlike Thrust, we advocate performing compaction directly in-kernel, as opposed to computing a prefix sum and scattering within a separate kernel. Fig. 1 illustrates the proposed In-Kernel Compaction in comparison to the Thrust scan-scatter operation.



(a) Out Of Kernel (b) InK-Compact

Figure 1: A parallel-operation is performed on the input and the resulting output is compacted (i.e. only contains valid elements), ready for further processing. (a) The widely used Thrust-Compact requires leaving the kernel and running a prefix-sum and scatter kernel. (b) The proposed In-Kernel Compaction performs the compaction within the operating kernel.

1.2. Previous Work

There are many applications of compaction in recent literature. The work by Dyken *et al.* [DZTS08] accelerated isosurface extraction using marching cubes. The work involves traversing a volume pyramid and compacting the active traversal paths for the next level of the tree. It is interesting in that not only does this use stream compaction, but also stream expansion where one thread can output many items.

Stream compaction was used to pack pixels ready for deferred shading in a coherent manner in the work by Hoberock [HLJH09]. Similarly, Garanzha et al. [GL10] used compaction and sorting to rearrange similar rays to maximize traversal coherence. The work was further developed to improve BVH construction [GPBG11].

Harris [HSO07] developed a parallel prefix sum (scan) for CUDA, allowing compaction to be performed quickly for general data-parallel GPU tasks. This implementation was included in the CudPP and Thrust library, and is widely used as the standard method for compaction and related operations. The newer library, Thrust, has become more favorable with researchers and consequently we focus on comparing

our compaction technique with the compaction method in the Thrust library.

Nobari [NLKB11] used the scan-scatter method by Horn [Hor05] to accelerate generation of random graphs from databases. Stream compaction was used in work by Hissoiny [HOBD11] to speed up dosimetric computations for radiotherapy, using Monte Carlo methods; specifically they compacted computations on photons that worked longer than others. Rather than having threads idle, computations on photons are limited to a user constant, after which the stream is compacted to remove completed items.

Schwarz [SS10] used compaction during voxelization of surfaces and solids. The work is notable for employing a multiple-kernel pipeline (to alleviate under-utilization of the GPU) where compaction is used to ensure good ordering of triangles ready for further processing. Tang et al. [TMLT11] employed stream compaction in kernel. In their method a fixed amount of space is reserved, then each block writes to its own private part of the total array. A second pass compacts the private arrays. This prefix sum can be executed as part of the second kernel. van Antwerpen [vA11] also incorporates the compaction within the kernel, but does not guarantee order preservation.

Billeter et al. [BOA09] suggested an approach that makes use of the popcount bit counter and masking on the bit-array to reduce workload by a factor of 32. However, they did not completely implement this algorithm, and thus no results were reported. In contrast, our InK-Compact method makes use of new functionality that allows each thread in a warp to know the predicate of all threads in the warp. More importantly, our approach is to operate compaction in the same kernel that is outputting a stream, i.e., we complete the compaction before leaving the kernel. This ensures that no memory needs to be written/cleared for invalid elements. Finally, novel use of new synchronization functions leads InK-Compact to be a simple and optimized compaction method. At the time of writing, we are unaware of any further developments with Billeter, *et al*'s research, nor with their implementation (Chag::PP) [BOA09].

For isosurface rendering, one approach is extraction via marching cubes [LC87] and rasterization of the resulting mesh. Wilhelms and Van Gelder [WVG92] employed a min-max octree for skipping empty cells, accelerating the extraction process. This approach was improved with several extensions, including view-dependent culling [LH98]. Sramek [Sra94] demonstrated direct ray casting of isosurfaces using a distance field to accelerate via ray jumping. Parker et al. [PPL*99] achieved interactive isosurface rendering from large volume data using a parallel ray tracer on a shared-memory supercomputer, employing a hierarchical grid acceleration structure. Similar implementations exploiting SIMD arithmetic and packet traversal achieved interactive performance on single desktops and workstations, using min-max kd-trees [WFM*05] and octrees [KWH09]. On the GPU, Hadwiger et al. [HSS*05] employed a multi-pass rasterization pipeline and an efficient secant solver for efficient isosurface ray casting. Hughes and Lim [HL09] employed an optimized min-max kd-tree traversal in CUDA, and achieve real-time rendering rates. The work also raised the issue of keeping an acceleration structure simple and rely more on ray stepping and texture caching. Gobbetti et al. [GMIG08] generate view and isovalue-dependent cuts of an octree out-of-core, then traverse the cut octree directly within a single-pass GPU shader. They achieve interactive framerates for a reduced gigavoxel data.

2. In-Kernel Stream Compaction

Modern GPGPU applications make use of Compute Languages (for example CUDA) that significantly simplify programming for massively-parallel systems. Code executes in parallel within *kernels*. Each kernel is divided up into *blocks* of *warps*, where each block is automatically (and independently) scheduled by the hardware to run on one of the many multi-processor cores. A *warp* is defined as a group of *threads* (typically 32) that operate at the same time on the hardware, i.e. they are implicitly synchronized at each instruction. For this work we assume each thread will have one input (e.g. pixel, ray, data-element), perform an action and produce an output. For a Kernel $\mathbf{K}$ we define it to have $\mathbf{B}$ number of blocks, where each block has $\mathbf{T}$ threads.

Stream compaction is the process of producing (in parallel) an output array $\mathbf{Y}_{[0\ldots M-1]}$, after an operation on $\mathbf{X}_{[0\ldots N-1]}$ inputs, of which only M elements are *valid*. In ray-tracing, for example, there will be M valid rays which hit geometry and only these M valid rays will need to be shaded. We typically define *valid* elements as those that pass a *predicate* test. For each valid element, the main challenge is determining the offset in the output array in relation to other valid elements. In other words an offset into the array is needed for each thread, which is equal to the number of prior threads with a valid element.

Conceptually, our InK-Compact method consists of three steps: computation of the thread offset $t^{(u)}$ within its warp, the warp offset $w^{(u)}$ within its block, and the block offset $b^{(u)}$ within its kernel. Our approach to per-warp prefix is the same as discussed in Billeter [BOA09] and Harris [Hwu11]. Unlike Harris [Hwu11], however, we use bit-decomposition and balloting to achieve the inter-warp scan, rather than use shared memory scan. Finally, our main original contribution is computing the block-offset through the use of block-sections, while maintaining the input-output data-ordering, and without leaving the operating kernel.

2.1. Thread Offset

Within a block, currently 32 threads are grouped together to make a warp. The threads in a warp run in lock-step with one another and special *warp-vote* functions are available to

them as a result. Warp-vote functions enable a single thread to find information about the other threads in the same warp. The most important addition to the CUDA framework for this work has been the `_ballot()` function, introduced in the Fermi architecture. We expect in time a similar function will be available to other compute languages.

Calling `_ballot()` with the thread's own predicate condition will enable each thread to know the predicate state of all $w^{(s)}$ threads in the warp. For CUDA 4.0, the warp size $w^{(s)}$ is 32 and the returned value of `_ballot()` is a 32-bit integer where the *bits* represent the predicates of the 32 threads. By masking the result of `_ballot()` such that bits representing threads after the current thread are set to zero, and counting the set-bits, we get the number of threads prior to the current thread (in the warp) that will write an output.

Firstly we define $w^{(i)} = \frac{t^{(i)}}{w^{(s)}}$ to calculate the warp index $w^{(i)}$ from the thread identifier number $t^{(i)}$, where $w^{(s)}$ is the size of a warp. We next define the thread's warp lane $w^{(l)} = t^{(i)}$ **mod** $w^{(s)}$, i.e. the thread index within warp. Finally, we compute the thread mask $t^{(m)} = \text{MAXINT}$ **shr** $(w^{(s)} - w^{(l)})$, which is a binary bit-mask with bit locations less-than $w^{(l)}$ set to 1 and all others set to 0. We define the number of warps within the block as $w^{(n)}$.

Next we can find the output-offset t^u for a thread within its warp as

$$\mathbf{b} = \text{ballot}(t^{(p)}) \textbf{ and } t^{(m)} \tag{1}$$

$$t^{(u)} = \text{popc}(\mathbf{b}) \tag{2}$$

The total number of writes within warp $w^{(i)}$ can be found by the last thread within it (i.e, where $w^{(l)} = w^{(s)} - 1$) as $w^{(c)} = t^{(u)} + p^{(t)}$, where the predicate $p^{(t)} \in [0,1]$. Finally, each warp $w^{(i)}$ stores its total $w^{(c)}$ within a shared memory array $\mathbf{w}^{(c)}_{w^{(i)}}$; performed by the last thread (in the warp) only. To allow other threads to read the values correctly a memory fence and a synchronisation is required.

2.2. Warp Offset

The next stage is to find the total number of writes prior to warp $w^{(i)}$, which serves as the output-offset $w^{(u)}$ for the warp. This is performed by a scan operation on all the warp counts $\mathbf{w}^{(c)}$. To ensure this information is visible for the next stage we perform a synchronization. By calling `_syncthreads_count()` and passing the predicate of each thread, we can also gather the total number of writes within the block, which will be used at a later stage.

A solution for computing the write offset $w^{(u)}_i$ for each warp, which also does not require more shared memory than the total number of warps, is to again use bit manipulation. The maximum possible value of $w^{(c)}_i$ is equal to $w^{(s)}$. This means that the problem can be decomposed into $\log_2(w^{(s)})$ separate binary-bit scan operations. Assuming the maximum number of warps equals the warp size, this operation can be performed by the first $w^{(n)}$ threads of a single warp.

The warp offset $w^{(u)}_i$, for warp $w^{(i)}$, is computed as

$$w^{(u)}_i = \sum_{j=0}^{\log_2(w^{(s)})} \text{popc}(b^{(j)} \textbf{ and } t^{(m)}) \textbf{ shr } i \tag{3}$$

$$\mathbf{b}^{(j)} = \text{ballot}(w^{(c)}_i \textbf{ and } 2^j) \tag{4}$$

With the thread offset within the warp and the warp offset within the block now known, the final problem is how to determine the offset within the kernel.

2.3. Order-Preserving Compaction

Achieving an in-kernel order-preserving compaction is challenging due to the various difficulties of block scheduling:

- While a kernel may have hundreds of blocks, only several may be active at the same time.
- There is no guarantee that blocks are activated linearly.
- A block must completely terminate before a new block can begin execution.

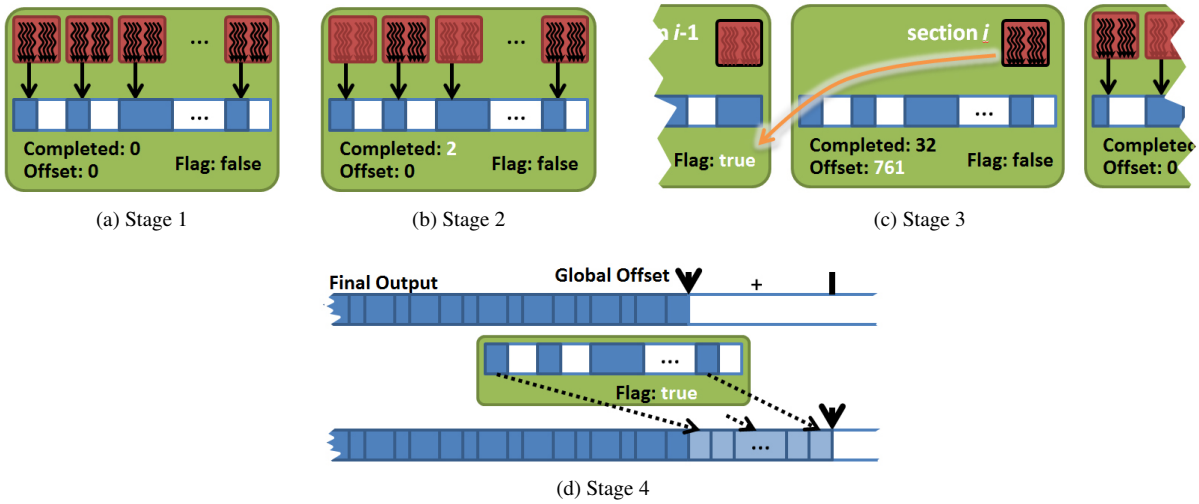
To achieve an order-preserving compaction under these difficulties, we use only guaranteed GPU behaviors outlined as follows. First, as a preprocess we divide a kernel into S sections, where each section represents $S^{(N)}$ blocks. A section size of $S^{(N)} = 32$ is chosen so that the final bit-decomposition can fit into a warp of 32-threads. Each section S_i has a boolean $S^{(f)}_i$, an offset value $S^{(u)}_i$ and counter value $S^{(c)}_i$. In addition we allocate an intermediate output buffer $\widehat{\mathbf{Y}}_{[0\ldots M-1]}$ and a buffer $\mathbf{B}^{(c)}$ to store the block counts. It is important that the arrays $S^{(f)}$ and $S^{(c)}$ are zero-cleared before the main kernel is executed.

2.3.1. Management of Sections in a Kernel

Here we describe how to manage the sections in a kernel to complete order-preserving compaction in four stages (see Fig. 2):

Stage 1: Each block in the section S_i performs a local-block compaction; the thread and warp offsets are computed. All valid thread outputs are then stored in the intermediate buffer $\widehat{\mathbf{Y}}$, offset by the block-id multiplied by the number of threads-per-block, plus the computed thread and warp offsets. In addition, the total number of writes, $k^{(c)}$, for the block is computed as a by-product of the initial thread-warp synchronisation.

Stage 2: A single thread in the block then atomically increments the section counter $S^{(c)}_i$ by one. All blocks within the section, except for the section controller block, are allowed to exit the compaction function. This behaviour allows the majority of blocks in the section to finish and



(a) Stage 1 (b) Stage 2 (c) Stage 3

(d) Stage 4

Figure 2: Overview of section management in a kernel. **(a)** ***Stage 1***. The blocks compact the thread data and output to an intermediate buffer, offset by $blockId \times blockDim$. **(b)** ***Stage 2***. When blocks complete, they atomically increment a *blocks–completed* counter. The last block (the section controller) waits for *blocks–completed* to equal the number of blocks in the section. All block counts are then loaded and a bit-wise prefix sum is performed to get the total number of writes in the section. **(c)** ***Stage 3***. Section S_i waits for the *section–completed* flag in section S_{i-1} to go true. **(d)** ***Stage 4***. The global offset is moved up by the section offset (number of writes in the section), the *section–completed* flag is set to true, which allows section S_{i+1} to proceed. Finally, the compacted data from the intermediate buffer for section i are moved to the final output array.

free up resources for other blocks to start commencing without delay. It also allows blocks to do additional work and/or additional compactions. The *section controller* is designated as being the block within the section that increments the section-counter $S_i^{(c)}$ to the number of blocks within a section $S^{(N)}$. The block-counts within the section are loaded into shared memory, by the section-controller, and a scan operation is performed to find the offset of each block. If we limit the number of blocks within a section to the warp size $w^{(s)}$, the scan operation can be performed by a single warp using bit-decomposition, as in Eq.3. The number of scans we must perform is $\log_2(w^{(s)}w^{(n)})$ and while we have implemented this in a single warp, there is no reason why this work could not be spread among the other warps. Through the scan operation we also compute the total number of output elements t.

Stage 3: The last block in section S_i waits for the flag $S_{i-1}^{(f)}$ in section S_{i-1} to go true. The first section S_0 progresses immediately to the next stage.

Stage 4: Once the total number of valid elements t is known for the section, we set $S_i^{(u)} = S_{i-1}^{(u)} + t$, wait for the memory to be set (using a memory fence operation) and set the flag $S_i^{(f)} = \text{true}$. The flag will then allow the next section S_{i+1} to proceed. Finally, the last block in section S_i moves the already block-compacted data from the intermediate output array $S_i^{(y)}$ to the final output array $\mathbf{Y}$.

2.4. In-Kernel Collation

It is commonly defined in literature that stream compaction will preserve the ordering of elements in the final output. However, *strictly* abiding the ordering constraint is not necessary for *some* algorithms.

Indeed the ordering of elements in the input or output arrays is contextual and spending time preserving the ordering may be wasteful especially if the resulting output is to be sorted. For example, a multi-kernel ray-casting pipeline might only require valid ray-indices as input and in theory it doesn't matter in which order, only that they are valid. In practice, rays that traverse the scene close to one another should be grouped together as much as possible, however, overall the ordering of the groups is not important. InK-Collate (In-Kernel Collation) is a compromise between ensuring local ordering within a block of threads, but not global ordering of kernel blocks. Within a block, the output location of $thread_N$ is guaranteed to be after $thread_{N-x}$ whereas the output location of $block_N$ is not guaranteed to be after that of $block_{N-x}$. However, the order of blocks will be generally close to each other for natural cache-hits to occur. This approach allows InK-Collate not to rely on other prior blocks to complete their output first.

Unlike InK-Compact, there is no intermediate process, rather the blocks immediately determine an output location in the final output as a kernel offset k^u. This kernel offset is

computed by a single thread within each block, as

$$k^{(u)} = \text{atomicAdd}(global\_counter, k^{c}), \quad \text{if } t = 0 \qquad (5)$$

where $k^{(u)}$ is the kernel offset stored in shared memory, $global\_counter$ is a single integer in global device memory and $k^{(c)}$ is calculated in Stage 1.

A memory fence and a synchronisation is required such that the array $\mathbf{s}^{(u)}$ and variable $k^{(u)}$ are visible to all threads in the block. Then the offset for each thread can be computed,

$$t^{f} = k^{(u)} + \mathbf{s}^{(u)}_{w^{(i)}} + t^{(u)}, \qquad (6)$$

where t^{f} is the final output location for the thread t, $k^{(u)}$ is the block offset, $\mathbf{s}^{(u)}_{w^{(i)}}$ is the warp offset and $t^{(u)}$ is the offset of the thread within the warp. The final process is to store the desired output into the final buffer at t^{f} and exit.

3. Multi-Kernel Isosurface Ray-casting Pipeline

Although it is possible to pack an entire rendering pipeline into a single kernel on current GPGPUs [HL09], it is still good practice to divide kernels up into smaller lightweight kernels to increase thread coherency and to achieve high performance optimization.

While this avoids having threads in different states due to large code branches, each kernel output must now be compacted. To demonstrate and stress-test our InK-Compact method we create a multiple-kernel rendering pipeline for ray casting iso-surfaces.

3.1. Rendering Pipeline Overview

Our per-frame isosurface ray-tracing pipeline consists of four kernels; ray creation, BVH traversal, leaf intersection, and shading. Each kernel benefits from compacted input, and/or needs to compact their output (see Fig. 3).

3.1.1. BVH Tree Setup

To accelerate isosurface ray-tracing of a volume, we employ a median-split BVH (balanced) tree, like the kd-tree by Wald [WFM*05] and Hughes [HL09]. By using a BVH tree rather than implicit splits, we can incorporate some simple tree-manipulation (condensation) with very little overhead. Overall, this results in simpler ray traversal and is more optimized for GPU application.

We found a leaf size of 16^3 voxels to be optimal due to good texture caching on the current GPUs. The min/max is found for each sub-volume and is uploaded to the GPU ready for use in the BVH update stage. The tree memory amounts to less than 2.3MB for a 512^3 8-bit volume with 16^3 leafs.

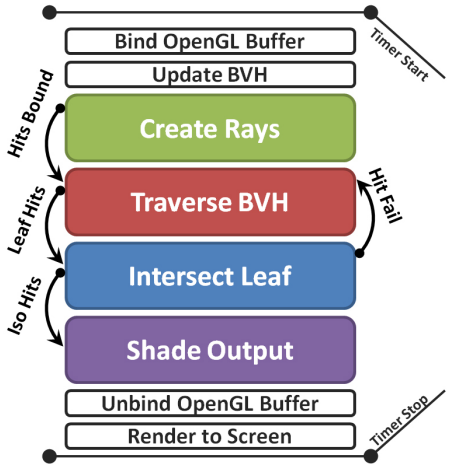
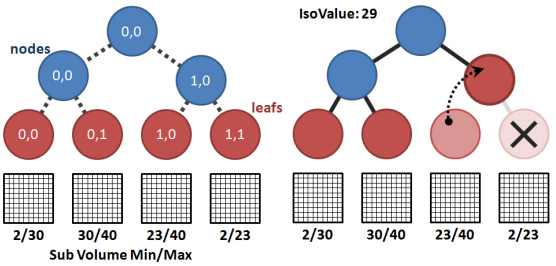


Figure 3: The pipeline of our per-frame multi-kernel isosurface ray-tracing. Each arrow represents a compaction of the work being fed into the next stage. Traversal and intersections kernels swap work back and forth several times.



(a) Implicit BVH Setup (b) Explicit BVH Update

Figure 4: Before rendering commences we first create a min/max sub-volume, with 16^3 leafs. We then create enough BVH nodes to create a balanced (implicit) BVH. During each frame, this implicit BVH is made into an explicit BVH specific to the current isovalue.

3.1.2. BVH Update Kernel

For each frame, we update the BVH tree to optimize it for the current isovalue. This firstly involves updating the leaves to test the isovalue against their min/max values available from the min/max sub-volume. If valid, a flag is set true in the node. The process is performed by multiple threads, one for each leaf. Fig. 4 illustrates this.

With the leaf validity flags set, we propagate up to the next level above to update the nodes. For each level, we determine the largest dimension and split along it. Subsequently, the child node offsets are computed; with an x-split, for example, they are $2x + [0,1], y, z$. If a parent node has no valid children, its flag is set to invalid. If the parent node has only one child, the details of the child are copied into the parent

(tree-compaction). Finally, if both children are valid, the parent's validity is set to true. During the node update, we also compute the bounds which encompass the child nodes.

3.1.3. Ray Generation Kernel

We create one ray per pixel and we immediately test it against the volume bound. We use stream compaction to remove any rays that do not intersect the volume at all. The compacted output is a list of ray indices ready for use in the first traversal kernel.

3.1.4. Traverse BVH Kernel

Traversal uses the standard BVH traversal method, where both children of the current node are loaded up and we test the ray against both axis-aligned bounding boxes. The child with the closest t_{near} is traversed into first, while the other child is pushed to the stack.

If the current node is a leaf, it is pushed to the stack, traversal stops and a call to the InK-Compact method is made. The traversal kernel has only one work output, which will be sent to the intersection kernel after compaction. Once the kernel completes and code returns to the CPU, the total number of ray indices in the work-list is accessed and we know how many threads will be needed.

3.1.5. Iso-Intersection Kernel

The intersection kernel takes an input list of indices of rays that need to perform an intersection test. The kernel execution will first access the ray information and pop the ray stack; this is a way of passing ray states from one kernel to another. The ray t_{near} and t_{far}values for the leaf bound are recomputed and then a loop commences to step along the ray incrementally. Once a crossing of the isovalue is detected, we perform linear-interpolation to find the intersection location. Finally, the normal is computed and stored. Additionally, the *output-to-shader* predicate is set.

Once it is known the Intersection-Kernel has failed to find an isosurface, we first peek the top of the ray stack to see if the next item is also a leaf. If so, we pop the stack and return the thread to the intersection loop. Doing the stack peek enables us to avoid the situation where the intersection kernel passes the ray to the traversal kernel, only for it to be returned back again to the intersection kernel. However, the drawback is that this can lead to many threads waiting a considerable amount of time while some threads retest for the new leaf. If intersection fails, the ray stack is not empty and the top stack item is not a leaf, then the *output-to-traversal* predicate is set.

Finally, two compactions are performed/required, firstly for the *output-to-shader* predicate and then for the *output-to-traversal*; What is actually performed here will be different, depending on the compaction method used.

3.2. Inter-Kernel Work Compaction

There are several compaction stages in our pipeline; Generation-Traversal, Traversal-Intersection, Intersection-Traversal, and Intersection-Shade.

When using our InK-Compact methods, the compaction is performed by calling either the InK-Compact or InK-Collate methods at the end of each kernel just prior to leaving the kernel. For the intersection kernel, this is performed twice for each output case.

When using Thrust and Chag::PP, however, the kernels will simply output the ray index to the output array in the same location as was read from the input array. The output array also requires non-valid elements be set to zero/null. Once the kernel completes, compaction is achieved by calling Thrust's copy-if method, or the Chag::PP.compact method, to compact (at most) the same number of input items.

Note that we *do not* wait for the kernels to complete (sync) before calling Thrust or Chag::PP, as this allows the CPU to add the compaction kernels to the CUDA work-list and minimize delays, thus allowing those approaches to achieve their optimal performance for comparison with ours.

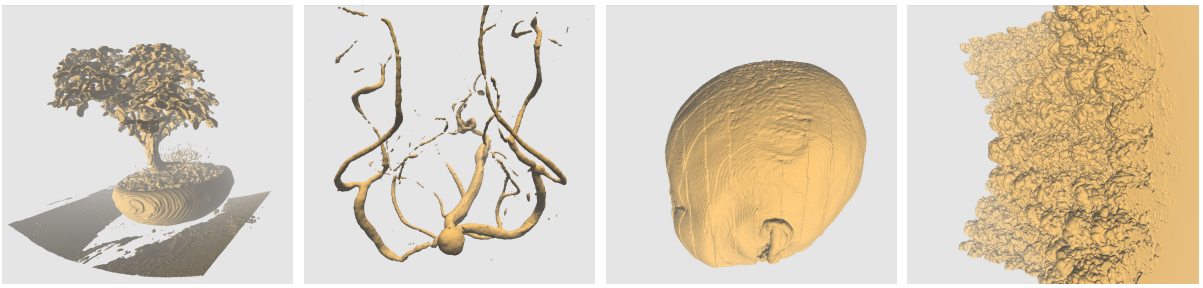
4. Results and Analysis

We have tested the proposed In-Kernel Compaction method (InK-Compact) and compared it against the widely used compaction methods of Thrust [HB10] and Chag:PP [BOA09], while using them in our multi-kernel isosurface ray-tracer. All tests were carried out on a standard NVidia 480 GTX graphics card. Furthermore, we compared our multiple-kernel rendering pipeline against a single-kernel pipeline. The single-kernel pipeline uses the same ray-tracing code as with the multiple-kernel pipeline, except that it has been split into three kernels and that there is additional work to initialise-from/output-to the compacted work-lists.

In our tests, we used four data sets which have different data characteristics and sizes (see Fig. 5). For example, the Bonsai and Aneurysm have thin features and large gaps between them; after high number of traversal-intersection steps, many rays end up hitting nothing. On the other hand, with the Head and LLNL data sets, most rays will definitely hit the surface after a small number of traversal-intersection steps.

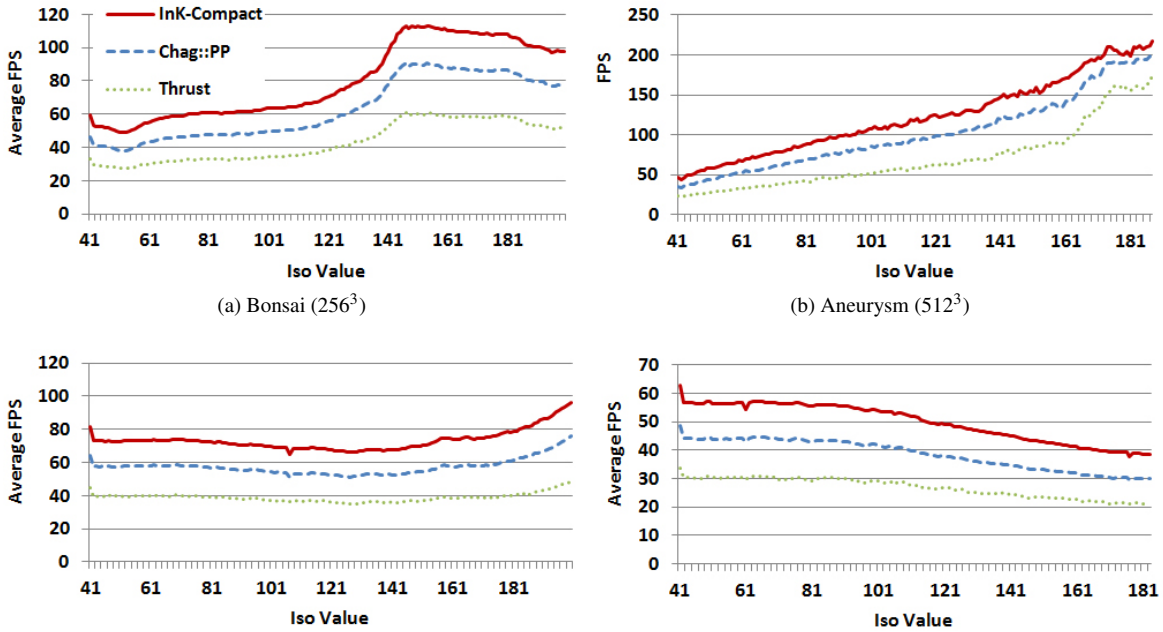
Fig. 6 shows frame rates for rendering each interesting (i.e., non-empty and little noisy) isosurface in the four data sets. The rendering resolution is set at 1024^2. The proposed InK-Compact consistently outperforms other compaction methods; on average, 90% and 28% improvements over Thrust and Chag::PP, respectively.

In this work, we assume that the iso-surface changes each frame regardless of whether it actually does and our rendering results presented in this paper include the cost of the



(a) Bonsai (256^3) (b) Aneurysm (512^3) (c) Head (256^3) (d) LLNL (512^3)

Figure 5: The test data sets which have different data characteristics and sizes.



(a) Bonsai (256^3) (b) Aneurysm (512^3)

(c) Head (256^3) (d) LLNL (512^3)

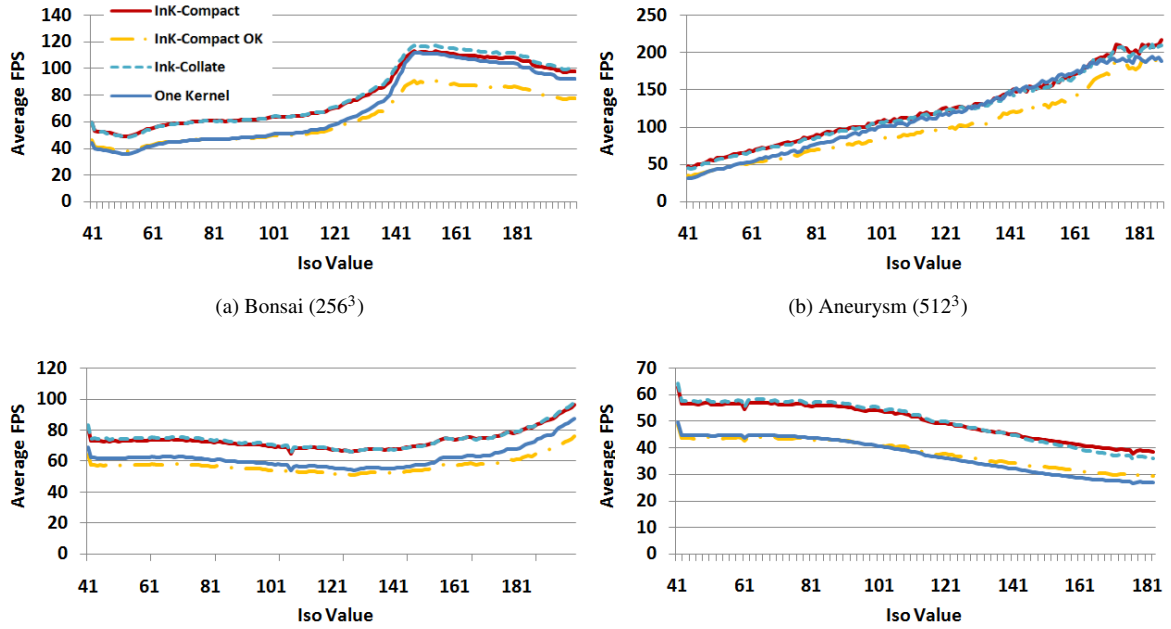
Figure 6: FPS (Frames Per Second) averaged over various view angles for each interesting (not-noisy/not-empty) isovalue; the higher, the better. The multi-kernel rendering pipeline using the proposed In-Kernel Compaction consistently outperforms the other alternatives based on conventional compaction methods. On average, our In-Kernel Methods are 99% faster than Thrust and 28% faster than Chag::PP when used in the ray-tracing pipeline tests.

BVH update. In practice, we could choose to only update the BVH upon isovalue changes and gain a minor speed boost.

Although it is designed for in-kernel compaction, we tested the out-of-kernel effectiveness of our InK-Compact as well. We performed this by wrapping the InK-Compact methods in a global kernel and calling the kernel when we wanted to compact data. We then applied this compaction method in the same code used for Thrust and Chag::PP. The results in Table 1 show that, when InK-Compact is used in an out-of-kernel fashion, it runs on par with Chag::PP. With this result we can surmise the main advantage of InK-Compact over Chag::PP and Thrust is that compaction and scatter operations are performed immediately within the operating kernels, which results in fewer bottlenecks. We do not require leaving the kernel after preparing temporary arrays and calling two or more kernels unlike other compaction methods such as Thrust and Chag::PP. By including the compaction routines within the kernel, there is a small register cost, but modern GPU architectures facilitate a larger amount of register space and it is not of concern. Note that it

Data	InK-Compact	OfK-Compact	Thrust	Chag::PP	Single-Kernel	Kd-Jump
Aneurysm (512^3)	**122.8** (±46.6)	100.8 (±44.4)	69.7 (±39.7)	101.9 (±39.7)	115.2 (±49.8)	52.1 (±20.0)
Bonsai (256^3)	**79.6** (±22.6)	62.7 (±18.6)	43.0 (±11.8)	63.0 (±18.4)	70.0 (±27.2)	48.0 (±18.2)
Head (256^3)	**73.2** (±6.0)	56.8 (±4.8)	38.7 (±2.5)	57.2 (±4.7)	61.7 (±6.8)	35.3 (±5.1)
LLNL (512^3)	**49.7** (±6.6)	38.1 (±5.2)	26.8 (±3.4)	38.5 (±5.2)	37.2 (±6.5)	27.2 (±4.1)

Table 1: Mean FPS (± Standard Deviation) for each data set using our In-Kernel Compaction (InK-Compact) and its Out-Of-Kernel implementation (OfK-Compact), compared against Thrust, Chag:PP, the Single-Kernel rendering pipeline and Kd-Jump [HL09]; which is also a single-kernel iso-surface ray caster. The proposed In-Kernel Compaction outperforms all the other alternatives of the multi-kernel or single-kernel rendering pipelines.



(a) Bonsai (256^3)

(b) Aneurysm (512^3)

(c) Head (256^3)

(d) LLNL (512^3)

Figure 7: Our In-Kernel Compaction used in the the Multi-Kernel ray-tracing pipeline, compared against the same ray-tracing code in a Single-Kernel pipeline. We also show our compact method used in the same fashion (used out-of-kernel) as Thrust and Chag::PP. See Fig. 8 for the rendered images at sample isovalues.

is not feasible to test Thrust or Chag::PP as In-Kernel since they are not designed nor implemented to run in kernel and require at least two or more additional kernel calls.

Our InK-Compact additionally reduces the overhead of compaction, so it encourages more use of compaction throughout the application pipeline, particularly encouraging multiple kernel applications. Consequently, this enables faster computation because it allows better concurrency on the GPU through the regular compaction. Our multi-kernel ray-tracing pipeline demonstrates this advantage when compared against the single-kernel pipeline that incorporated the same ray-tracing code; on average, 15% speed-up over the single-kernel (see Table 1).

The multi-kernel ray-tracing based on the proposed InK-Compact outperforms the single-kernel ray-tracing at most isovalues, especially when the isosurfaces are complex. Only when the isosurface becomes simple, i.e., little to be rendered, does the single-kernel outperform the multi-kernel; because there is so little work (the majority of rays terminate quickly) and the CPU-side overhead for the multi-kernel becomes a burden. However, even in these cases, the multi-kernel is highly competitive, returning well over 100 fps (see Fig. 7 and Fig. 8). The overall average of 15% speed-up is thus the under-estimate in the sense that higher speed-ups are obtained for the complex isosurfaces which are computationally more demanding to render. In Fig. 7 we additionally provide a comparison for our non-order preserving In Ker-

nal Collation (InK-Collate) which for this ray-tracing application shows little difference from InK-Compact.

5. Limitations

Our InK-Compact is designed to run as a device function of an operation kernel and as such causes a register burden. However, with recent changes to GPU architecture (Fermi for example), this burden is not great. For example, once the main work of a kernel is complete and it moves onto the compaction, those registers used in the main work are no longer required and are reallocated for the compaction code.

Another possible limitation is the fact that we require an intermediate buffer in order to store block outputs, which allows blocks to exit and not lock up GPU resources. In theory, if the data type being compacted is large, this intermediate buffer would cause more memory usage than other compaction libraries. Our method will use a memory amount equivalent to the number of elements multiplied by the element size, whereas Chag::PP avoids this by storing a small prefix sum with one element per warp.

6. Future Work

A logical extension of this work is to combine in-core compaction with out-of-core reduction techniques. With careful design, concurrent kernel access can be achieved to better increase GPU occupancy at all times. It would also be interesting to trial more applications for In-Kernel Compaction. Mesh extraction and geometric analysis, for example, could be of interest. Finally, our immediate goal is to expand the multi-kernel rendering pipeline to incorporate mixed primitives, where specialized kernels handling different overlapping geometries would require heavy use of stream compaction.

Another interesting question is how to organise the workload in-between kernels. In our pipeline we used ray indices as it seems logical to keep the amount of memory transactions per-frame to a minimum. However, eventually memory access to gather the actual ray information will be incoherent. An interesting hypothesis is that by actually reorganizing the ray information itself, per iteration, all memory access will be coherent and would offset the extra cost. However, such a system would only be feasible with stackless traversal.

7. Conclusion

We have proposed InK-Compact, a novel in-kernel order-preserving compaction method for GPGPU applications. We have also presented a multi-kernel ray-tracing pipeline that makes effective use of stream compaction to organise ray work in between kernel calls. The proposed InK-Compact approach consistently outperforms the multi-kernel ray-tracing pipelines based on the state-of-art compaction methods such as Thrust and Chag:PP. Our multi-kernel ray-tracing pipeline based on InK-Compact also outperforms a single-kernel rendering pipeline except for simple cases of mostly empty rendering. We believe that many computationally intensive GPGPU applications (in addition to computer graphics and visualization) requiring compaction, especially those at multiple stages and those with multiple outputs, will benefit from the proposed In-Kernel Stream Compaction method.

8. Acknowledgments

The work presented in this paper was supported by RIVIC (the Wales Research Institute of Visual Computing) funded by HEFCW (Higher Education Funding Council for Wales).

References

[BOA09] BILLETER M., OLSSON O., ASSARSSON U.: Efficient stream compaction on wide SIMD many-core architectures. In *Proceedings of the Conference on High Performance Graphics* (2009), ACM, pp. 159–166. 2, 3, 7

[DZTS08] DYKEN C., ZIEGLER G., THEOBALT C., SEIDEL H.: High-speed Marching Cubes using HistoPyramids. *Computer Graphics Forum 27*, 8 (2008), 2028–2039. 2

[GL10] GARANZHA K., LOOP C.: Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum 29*, 2 (2010), 289–298. 2

[GMIG08] GOBBETTI E., MARTON F., IGLESIAS GUITIÁN J.: A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer 24*, 7 (2008), 797–806. 3

[GPBG11] GARANZHA K., PREMOZE S., BELY A., GALAKTIONOV V.: Grid-based SAH BVH construction on a GPU. *Vis. Comput. 27*, 6-8 (June 2011), 697–706. 2

[HB10] HOBEROCK J., BELL N.: Thrust: A parallel template library, 2010. Version 1.3.0. 2, 7

[HL09] HUGHES D. M., LIM I. S.: Kd-Jump: a Path-Preserving Stackless Traversal for Faster Isosurface Raytracing on GPUs. *IEEE Transactions on Visualization and Computer Graphics 15* (2009), 1555–1562. 2, 3, 6, 9

[HLJH09] HOBEROCK J., LU V., JIA Y., HART J.: Stream compaction for deferred shading. In *Proceedings of the Conference on High Performance Graphics* (2009), pp. 173–180. 2

[HOBD11] HISSOINY S., OZELL B., BOUCHARD H., DESPRÉS P.: GPUMCD: a new GPU-oriented Monte Carlo dose calculation platform. arXiv:1101.1245v1 [physics.med-ph], 2011. 3

[Hor05] HORN D.: Stream reduction operations for GPGPU applications. In *GPU Gems 2* (2005), Addison-Wesley, pp. 573–589. 3

[HS86] HILLIS W. D., STEELE JR. G. L.: Data parallel algorithms. *Communications of the ACM 29*, 12 (1986), 1170–1183. 1

[HSO07] HARRIS M., SENGUPTA S., OWENS J.: Parallel prefix sum (scan) with CUDA. In *GPU Gems 3* (2007), pp. 851–876. 1, 2

[HSS*05] HADWIGER M., SIGG C., SCHARSACH H., BÜHLER K., GROSS M.: Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum 24*, 3 (2005), 303–312. 3

[Hwu11] HWU W.-M. W.: *GPU Computing Gems Emerald Edition*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011. 3

[KWH09] KNOLL A., WALD I., HANSEN C.: Coherent Multiresolution Isosurface Ray Tracing. *The Visual Computer 25*, 3 (2009), 209–225. 3

[LC87] LORENSEN W., CLINE H.: Marching cubes: A high resolution 3D surface construction algorithm. *ACM Siggraph Computer Graphics 21*, 4 (1987), 163–169. 3

[LH98] LIVNAT Y., HANSEN C. D.: View dependent isosurface extraction. In *Proceedings of IEEE Visualization 1998* (1998), pp. 175–180. 3

[NLKB11] NOBARI S., LU X., KARRAS P., BRESSAN S.: Fast random graph generation. In *Proceedings of the 14th International Conference on Extending Database Technology* (New York, NY, USA, 2011), ACM, pp. 331–342. 3

[PPL*99] PARKER S., PARKER M., LIVNAT Y., SLOAN P.-P., HANSEN C., SHIRLEY P.: Interactive Ray Tracing for Volume Visualization. *Computer Graphics and Applications 5*, 3 (1999), 238–250. 3

[SHG09] SATISH N., HARRIS M., GARLAND M.: Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing* (2009), pp. 1–10. 1

[SHZO07] SENGUPTA S., HARRIS M., ZHANG Y., OWENS J. D.: Scan Primitives for GPU Computing. In *Proceedings of Symposium on Graphics Hardware* (2007), pp. 97–106. 1, 2

[Sra94] SRAMEK M.: Fast Surface Rendering from Raster Data by Voxel Traversal Using Chessboard Distance. *Proceedings of IEEE Visualization 1994* (1994), 188–195. 3

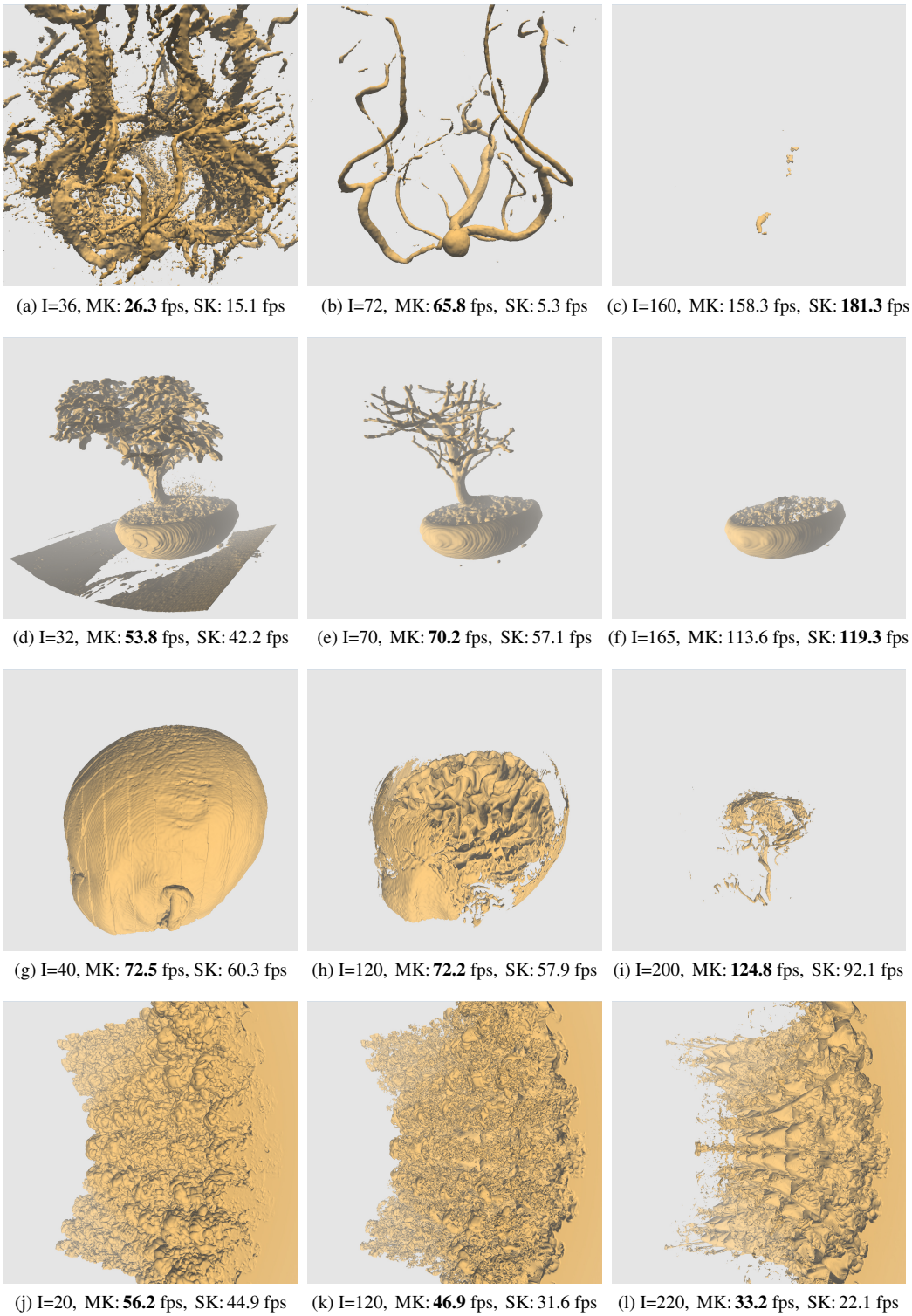
[SS10] SCHWARZ M., SEIDEL H.: Fast parallel surface and solid voxelization on GPUs. *ACM Transactions on Graphics 29*, 6 (2010), 179(1–9). 3

[TMLT11] TANG M., MANOCHA D., LIN J., TONG R.: Collision-streams: fast gpu-based collision detection for deformable models. In *Symposium on Interactive 3D Graphics and Games* (2011), I3D '11, pp. 63–70. 3

[vA11] VAN ANTWERPEN D.: Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU. In *High-Performance Graphics 2011* (2011), Dachsbacher C., Mark W., Pantaleoni J., (Eds.), pp. 41–50. 3

[WFM*05] WALD I., FRIEDRICH H., MARMITT G., SLUSALLEK P., SEIDEL H.-P.: Faster Isosurface Ray Tracing Using Implicit KD-Trees. *IEEE Transactions on Visualization and Computer Graphics 11*, 5 (2005), 562–573. 3, 6

[WVG92] WILHELMS J., VAN GELDER A.: Octrees for faster isosurface generation. *ACM Transactions on Graphics (TOG) 11*, 3 (1992), 201–227. 3



(a) I=36, MK: **26.3** fps, SK: 15.1 fps (b) I=72, MK: **65.8** fps, SK: 5.3 fps (c) I=160, MK: 158.3 fps, SK: **181.3** fps

(d) I=32, MK: **53.8** fps, SK: 42.2 fps (e) I=70, MK: **70.2** fps, SK: 57.1 fps (f) I=165, MK: 113.6 fps, SK: **119.3** fps

(g) I=40, MK: **72.5** fps, SK: 60.3 fps (h) I=120, MK: **72.2** fps, SK: 57.9 fps (i) I=200, MK: **124.8** fps, SK: 92.1 fps

(j) I=20, MK: **56.2** fps, SK: 44.9 fps (k) I=120, MK: **46.9** fps, SK: 31.6 fps (l) I=220, MK: **33.2** fps, SK: 22.1 fps

Figure 8: Comparison of Multi-Kernel vs. Single-Kernel ray-tracing pipelines at various isovalues. The multi-kernel ray-tracing based on the proposed In-Kernel Compact outperforms the single-kernel ray-tracing at most isovalues, especially when the isosurfaces are complex. Only when the isosurfaces become simple, i.e., little to be rendered as in (c) and (f), the single-kernel outperforms the multi-kernel; even in this case, however, the multi-kernel is highly competitive, returning well over 100 fps.