Data-Dependent Multipass Control Flow on GPUs

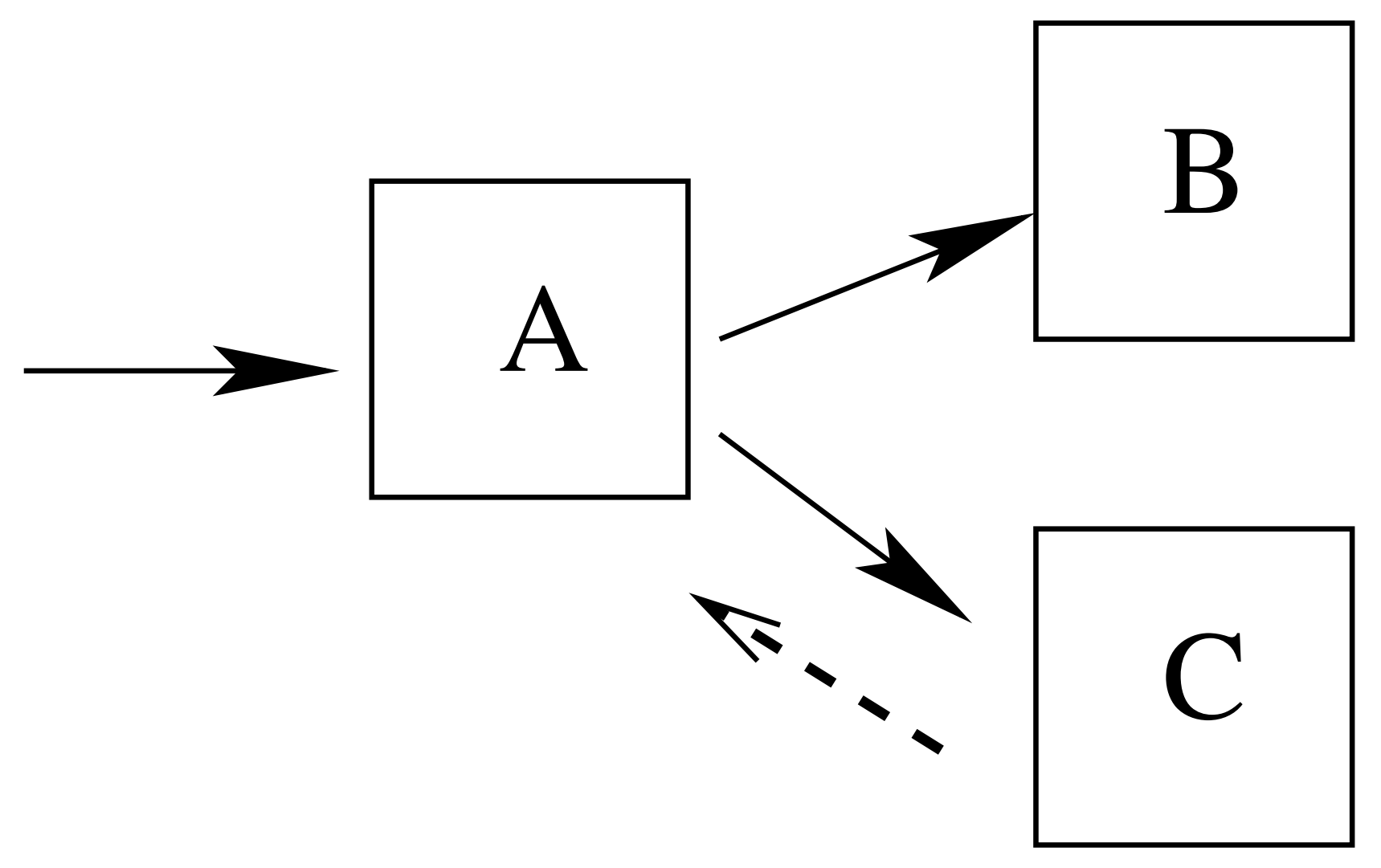
Tiberiu Popa ⋆ Michael McCool
Computer Graphics Lab, School of Computer Science, University of Waterloo

The computations performed by GPUs are suitable for a streaming computational model [3], and GPU architectures are similar to those of stream processors. However, SIMD GPUs do not efficiently implement data-dependent control flow.

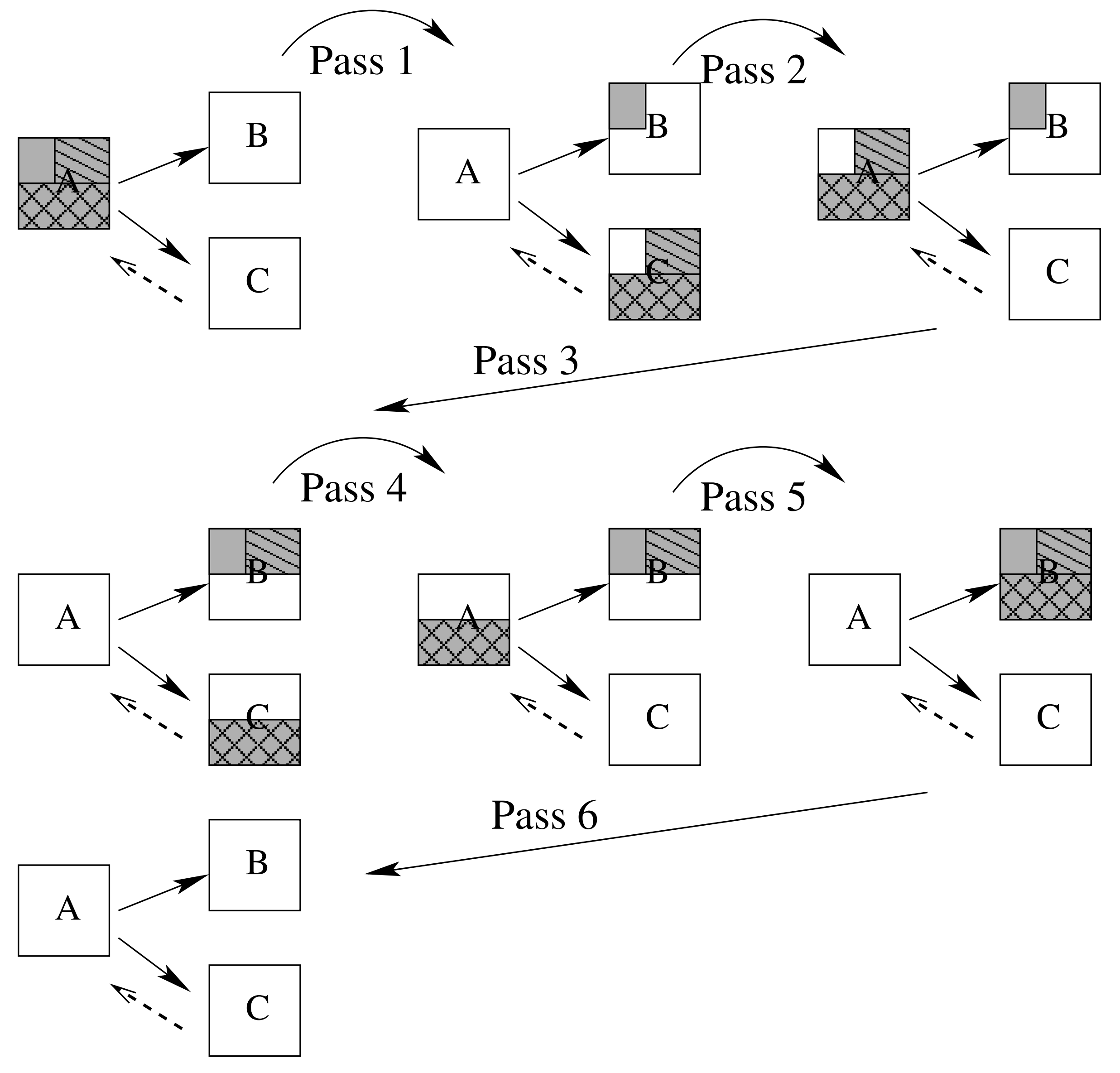
We are attempting to build an efficient, portable, and transparent multipass programming framework that supports the entire palette of data-dependent control constructs. For our prototype we restricted our work to GPU fragment units. Streams are stored in textures for reading operations and target frame buffers for writing operations. However, on current GPUs fragments are written to an array, not sequentially to a stream. In particular, null records (killed fragments) resulting from conditional outputs are not removed from the output stream resulting in wasted bandwidth and computation on later passes. Kapasi et al [1] use special hardware to pack the stream into a compact representation with no null records. The compacted stream can then be processed in later passes with maximum efficiency. We are attempting to apply this approach to GPUs.

Scheduler

The control flow of a program can be schematically represented as a directed graph where the nodes have only linear control flow and the arcs represent data dependent branches. In a multipass streaming computation framework, the control graph can be interpreted as a *streaming graph*. Nodes are independent *kernels* and arcs are data paths from one kernel to the next. For example, the following figure shows the streaming graph of a Julia set test program. This streaming graph consists of three kernels, representing initialization code, the body of the loop and the evaluation of the final colour. Its data flow graph is illustrated in the following figure. We execute the streaming graph in multiple passes, packing data into different buffers to eliminate garbage computations. In each pass, a kernel will be selected by the scheduler based on heuristics to maximize throughput.

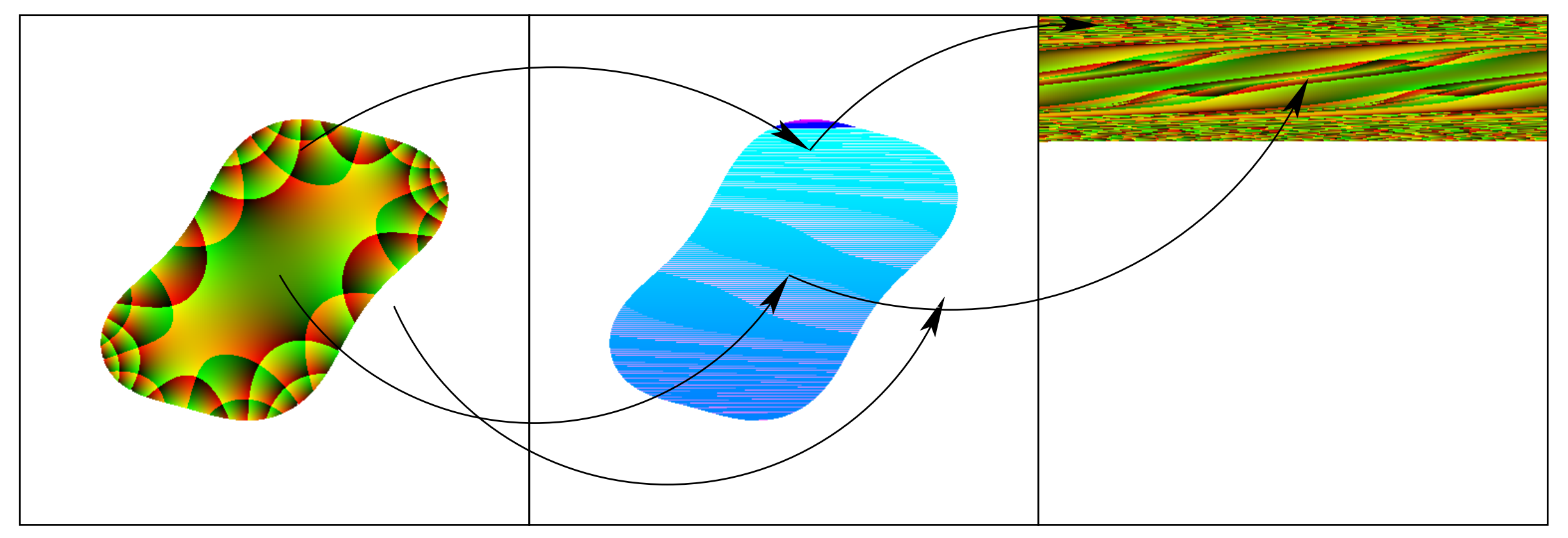


Our system takes a general purpose program written in Sh [2], and transforms its control graph into a streaming graph suitable for multi-pass processing. A run-time scheduler then searches for a suitable sequence in which to run the kernels. The following figure demonstrates the execution of a streaming graph.



Branches are handled using conditional assignment and multiple output buffers. As a result, output streams are split into multiple sparse streams. To avoid resource idling, we explicitly perform a packing operation on sparse arrays to convert them to dense arrays.

Packing



Packing is the spatial compaction of stream data. Unpacking is the opposite operation that merges two previously packed streams into one stream. The above figure shows how a sparse stream is packed and can later be unpacked using an inverse map stored in a texture. The left image shows a sparse stream, the right image shows the packed stream and the middle image shows the inverse map used to restore the order of the stream.

Julia Set Example

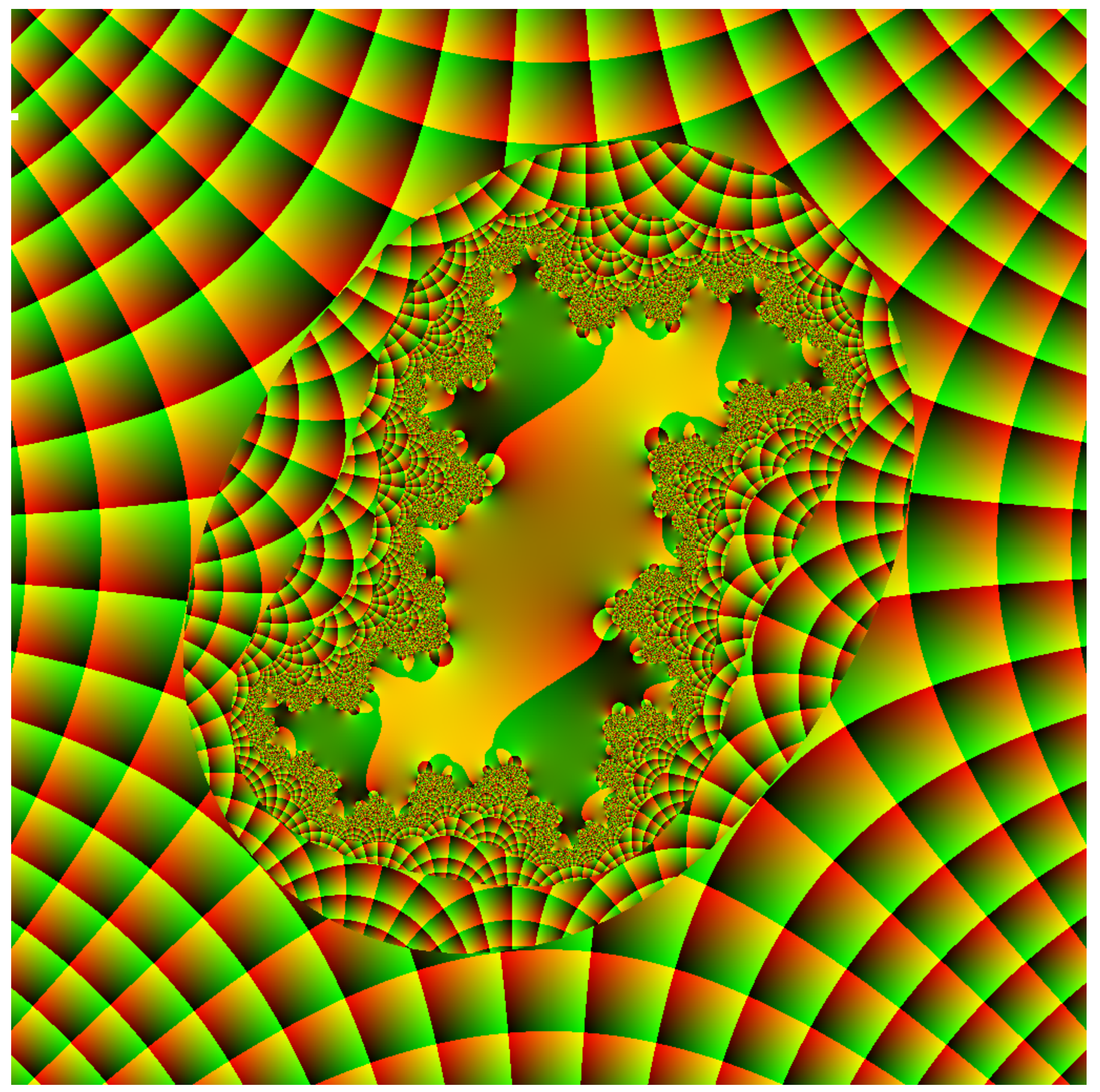
The potential improvement in performance on the Julia set test case is given in the following table:

Maximum iterations	Improvement incl. overhead	Improvement w/o overhead
5	22%	35%
10	26%	40%
20	28%	41%

Julia set evaluation stresses iteration scheduling due to its high iteration count variance. The performance gain for computing the Julia set using packing over not packing is illustrated in the above table. The resolution was 512×512. The second column shows the overall gain in performance (including scheduling and kernel-switching overhead) and the third column shows the gain in fragment program execution efficiency alone.

These numbers do not take into consideration the cost of packing. Unfortunately, packing on current GPUs is relatively expensive. However, it would be possible to add hardware support to GPUs to support free packing on write, or low-cost packing during blit operations.

```
ShAttrib1f iterations =  5.0; // number of iterations
ShAttrib1f color_scale_factor =  1.0 / (iterations+1.0);
ShAttrib3f c(-0.122, 0.745, 0.0); // julia set constant

ShProgram ifp = SH_BEGIN_PROGRAM("gpu:stream") {
  ShInputColor3f input;     // (x, y) position of a point
                            // in the interval [-2, 2]
  ShOutputColor3f output;   // color

  ShAttrib3f pos = input;   // stores the positions
  ShAttrib3f i(0, 0, 0);    // iterator variable

  // first iteration
  ShAttrib3f temp = pos;
  pos(0) = temp(0) * temp(0) - temp(1) * temp(1) + c(0);
  pos(1) = 2.0 * temp(0) * temp(1) + c(1);

  // stopping conditions
  SH_WHILE( (i(0)<iterations) *
           (pos(0) * pos(0) + pos(1) * pos(1) <= 4.0f)) {
    // iterate
    ShAttrib3f temp = pos;
    pos(0) = temp(0) * temp(0) - temp(1) * temp(1) + c(0);
    pos(1) = 2.0 * temp(0) * temp(1) + c(1);

    i(0) = i(0) + 1;
  } SH_ENDWHILE;

  // final colour
  output = pos;
} SH_END;
```



Conclusion

We have presented a method to implement asymptotically efficient data-dependent control flow on SIMD GPUs using conditional stream output and packing. A run-time scheduling algorithm determines the order in which kernels run. Sparse stream elements are packed into contiguous blocks to avoid redundant computations.

On current GPUs, the overhead associated with packing is large, so hardware support would be required to make this approach practical. However, similar alternative approaches to avoid computation are possible; for instance, by exploiting vertex buffer feedback, the occlusion test, and early depth testing.

References

[1] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. Efficient conditional operations for data-parallel architectures. In *Symp. Microarchitecture*, pages 159–170. ACM Press, 2000.

[2] Michael McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh*. AK Peters Ltd., 2004.

[3] John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon Rendering on a Stream Architecture. *Proc. Graphics Hardware*, pages 23–32, 2000.