

The Graphics Card as a Stream Computer

Suresh Venkatasubramanian
AT&T Labs – Research

1 Introduction

Massive data sets have radically changed our understanding of how to design efficient algorithms; the *streaming paradigm*, whether it in terms of number of passes of an external memory algorithm, or the single pass and limited memory of a stream algorithm, appears to be the dominant method for coping with large data.

A very different kind of massive computation has had the same effect at the level of the CPU. It has long been observed [?] that the traditional Von Neumann-style architecture creates memory bottlenecks between the CPU and main memory, and much of chip design in the past many years has been focused on methods to alleviate this bottleneck, by way of fast memory, caches, prefetching strategies and the like. However, all of this has made the memory bottleneck itself the focus of chip optimization efforts, and has reflected in the amount of real estate on a chip devoted to caching and memory access circuitry, as compared to the ALU itself [?].

For compute-intensive operations, this is an unacceptable tradeoff. The most prominent example is that of the computations performed by a graphics card. The operations themselves are very simple, and require very little memory, but require the ability to perform many computations extremely fast and in parallel to whatever degree possible. Inspired in part by dataflow architectures and systolic arrays, the development of graphics chips focused on high computation throughput while sacrificing (to a degree) the generality of a CPU.

What has resulted is a *stream processor* that is highly optimized for stream computations. Today's GPUs (graphics processing units) can process over 50 million triangles and 4 billion pixels in one second. Their "Moore's Law" is faster than that for CPUs, owing primarily to their stream architecture which enables all additional transistors to be devoted to increasing computational power directly. An intriguing side effect of this is the growing use of a graphics card as a general purpose stream processing engine. In an ever-increasing array of applications, researchers are discovering that performing a computation on a graphics card is far faster than performing it on a CPU, and so are using a GPU as a stream co-processor. Another feature that makes the graphics pipeline attractive (and distinguishes it from other stream architectures) is the *spatial parallelism* it provides. Conceptually, each pixel on the screen can be viewed as a stream processor, potentially giving a large degree of parallelism.

2 A Model Of A Graphics Card

A detailed description of the architecture of a graphics card is beyond the scope of this note; what follows is a distillation of the key components. The stream elements that a card processes are points, and at a later stage, *fragments*, which are essentially pixels with color and depth information. A program issues requests to the card by specifying points, lines,

or facets, (and their attributes) and issuing a rendering request. The vertices go thru several processing phases (called a *vertex program*) and finally are scan-converted into fragments by a *rasterizer* that takes a three-dimensional scene and breaks it into pixels. Each fragment is then processed by another stream computation (called a *fragment program*) and finally ends up as a pixel on a screen. Alternatively, the pixels and their attributes can be captured in internal storage and their contents can be retrieved to the CPU. Spatial parallelism occurs at the level of a fragment; each fragment program can be viewed as running in parallel at each pixel. The vertex programs are executed on each point. *Texture memory* provides a form of limited local storage for maintaining state; access to it is restricted though.

2.1 A Computational Model

From the above description, it is clear that a GPU executes a stream program. There are some key differences between it and a general purpose stream program. The most essential one is that *every object in the stream is processed by the same function*.

Recall that in a general stream computation [?] the data streams by an *arbitrary* Turing machine that has access to limited memory. There is no restriction in general on what is computed; the restriction that the hardware enforces is thus significant. Further, the computations themselves are limited to branching programs; no looping or jumping is allowed¹.

Another important (and related) difference is the access to memory; a stream program on a graphics card has access to much smaller memory than a general stream program. Although graphics cards have a great deal of memory on them (256 MB in the best cards available today), each processing unit really has access to only a few fast registers, unlike the much larger memory limits on a standard streaming process.

Finally, the typical computation performed in the GPU is multi-pass; a constant or even $\omega(1)$ passes may be required for a particular computation.

2.2 Less Is More

Although it seems that the above restrictions are artificial and limiting, it turns out they are at the heart of the performance gains achieved by graphics cards. Because each item in the stream is processed in the same way, microcode for the computation can be loaded directly onto the GPU, and no expensive memory access are needed to fetch the next instruction. Using branching programs enforces the pipeline discipline, and ensures that items are processed without stalls. Lack of memory access allows small caches to be used and prevents the memory bottleneck from swamping the chip. It is also important to observe that other streaming architectures that have been designed and proposed ([?; ?]) share little in common with graphics cards but share the

¹there are some exceptions, but these are not significant

above *uniformity* assumption. Thus, uniform streaming appears to be the common underlying computational metaphor for stream architectures in general.

Limitations On Computing Power As far back as 1989, Fournier and Fussell [?] presented a view of the GPU as a stream computing engine, and presented various upper and lower bounds for simple problems. They show for example that sorting is a hard problem in this model, and investigate how the power of the pipeline changes as more and more registers and operations are added to the model.

More recently, Guha *et al.* [?] showed that computing the median of n numbers takes n passes in the “standard” graphics pipeline model. This contrasts to $O(\log n)$ passes required to compute the median in a stream model. Interestingly, the addition of an extra operator (an interval test of the form “Given z , is $a \leq z \leq b$ ”) reduces the number of passes needed (randomized) to $O(\log n)$.

3 Examples

There are numerous examples of places where the GPU has been used to perform general stream computations. A somewhat incomplete list is below:

- General distance field calculations: Given a distance function and a set of objects, compute for each object its (nearest/farthest/median) neighbour. Yields algorithms for Voronoi diagrams [?], general shape fitting [?], spatial convolution and outlier detection and other problems.
- Object intersections: given two objects, determine if they intersect and by how much. Yields algorithms for spatial joins [?], penetration depth [?], range searching [?].
- Scientific computing: Solve partial differential equation via finite element methods [?]
- Physical simulation: Simulate the motion of objects in a physically realistic way. Useful for game programming and dynamical system modelling.

It is worth noting that none of the above applications make use of the extensive features of vertex and fragment programs, and in fact only use a very primitive subset of operations.

4 Directions

There are a number of challenging directions to pursue. There is already work on software and language constructs to express general purpose stream programs in terms of low-level graphics hardware commands (so that the programmer need know nothing about graphics) [?]. There is work in compilers on optimizing the translation of high-level stream constructs [?]. Other work seeks to extend the uniform streaming model to allow limited caching of state [?].

From a computational standpoint, all of this leads to a fundamental question: What is the power of uniform streaming and its variants? The answer will be invaluable not only from a theoretical point of view, but as a way of indicating what problems are likely to be amenable to efficient stream computation.