

Compiling Data Dependent Control Flow on SIMD GPUs

by

Tiberiu S. Popa

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2004

©Tiberiu S. Popa 2004

Author's Declaration for Electronic Submission of a Thesis

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Current Graphic Processing Units (GPUs) (circa. 2003/2004) have programmable vertex and fragment units. Often these units are implemented as SIMD processors employing parallel pipelines. Data dependent conditional execution on SIMD architectures implemented using processor idling is inefficient.

I propose a multi-pass approach based on conditional streams which allows dynamic load balancing of the fragment units of the GPU and better theoretical performance on programs using data dependent conditionals and loops. The proposed system can be used to turn the fragment unit of a SIMD GPU into a stream processor with data dependent control flow.

Acknowledgements

I would like to thank my supervisor Michael McCool for his guidance, expertise and support. His energy and dedication to his students was an inspiration.

I would like to thank my readers Stephen Mann and Peter Buhr for taking the time to read my thesis and for their useful suggestions.

I would like to thank all the members of the Computer Graphics Lab for making the lab such a unique interactive environment, and especially to Kevin Moule who was the best mentor I could hope for.

I would also like to thank my wife Paula Popa for her love and devotion in coping with my graduate lifestyle and to my parents Anca and Stelian Popa for their support, encouragement, confidence and belief in me.

This thesis was partially funded by Ontario Graduate Scholarship (OGS), National Science and Engineering Research Council of Canada (NSERC) and Electronic Arts.

Contents

1	Introduction	1
2	Background	6
2.1	Parallel Architectures	6
2.1.1	Single Instruction Stream Multiple Data Streams (SIMD)	7
2.1.2	Stream Processing Model	9
2.2	GPU Architectures	13
2.3	GPUs and Stream Processing	17
2.4	Parallel Performance	20
2.5	Sh System	22
2.5.1	Stream Abstraction	22
2.5.2	Shader Algebra	23
2.6	Organizational Notes	25
3	The Sm System	27
3.1	The Sm Simulator	27
3.2	Scheduling Strategies	31

3.3	Simulator Results	33
3.4	Performance Analysis	33
4	GPU System	41
4.1	Stream Packing and Unpacking	42
4.1.1	Stream Packing using Beneš Networks	42
4.1.2	Stream Packing on Current GPUs	44
4.1.3	Unpacking	45
4.2	Graph Structure	49
4.2.1	Node Types	50
4.2.2	Graph Transformation Algorithm	57
4.3	Examples	62
4.3.1	Checkerboard	62
4.3.2	Computing the Julia set	64
4.3.3	Combined Example	66
4.4	Scheduler	70
4.4.1	Scheduling Algorithm	70
4.4.2	Scheduling Strategies	73
4.5	Performance Analysis	75
4.5.1	Cost Models	75
4.5.2	Methodology	76
4.5.3	Benchmarks	77
5	Conclusion	81

5.1	Systems Comparison	82
5.2	Future Work	82
	Bibliography	85

List of Tables

3.1	Sm Scheduler Performance	36
3.2	Sm Streaming Graph Symbols	37
3.3	Sm Kernel Switches Analysis	38
4.1	Active Node Conditions	70
4.2	Stream Performance	78
4.3	Packing Performance	80
4.4	Work Distribution	80

List of Figures

1.1	Conditional Data Flow	4
2.1	SIMD Execution	8
2.2	Strip Mining	9
2.3	Stream Partitioning and Compression	11
2.4	Switching and Combining Conditional Streams	12
2.5	Simplified GPU Pipeline	14
2.6	Conditional Flow	18
2.7	Synchronous Execution	21
2.8	Asynchronous Execution	21
2.9	Algebra Ambiguities	25
2.10	Stream Storage	26
2.11	Stream Containers Notation	26
3.1	Example of a Stack Arc	29
3.2	Greedy Kernel Scheduling	32
3.3	Test Cases	34
3.4	Sm Streaming Graphs	39

4.1	Sorting Network	43
4.2	Packing and Unpacking	47
4.3	Resolving Data dependencies	48
4.4	Stream Registers Fill	51
4.5	Unpacked Frame Buffer Fill	52
4.6	Packed Frame Buffer Fill	52
4.7	Non-synchronized Scheduling for Conditionals	53
4.8	Synchronized Scheduling for Conditionals	54
4.9	Non-synchronized Scheduling for Iterations	55
4.10	Synchronized Scheduling for Iterations	56
4.11	Compiler Graphs	58
4.12	Auxiliary Programs	60
4.13	Checkerboard	62
4.14	Julia Set Control Graph	66
4.15	Julia Set Output	67
4.16	Checkerboarded Julia Set	67
4.17	Checkerboarded Julia Set Streaming Graph	69
4.18	Deadlock Avoidance	72
4.19	Merging Packed Streams	74

Chapter 1

Introduction

Historical data shows that the performance of Graphics Processing Units (GPUs) has been increasing significantly faster than that of Central Processing Units (CPUs) [11]. Both types of processing units are subject to the same advances in semi-conductor technology, so the growing gap in performance can be attributed to the difference between their computational models. Some GPUs are Single Instruction Stream Multiple Data Streams (SIMD) machines, employing a stream computing model that maps well onto hardware, particularly with respect to coherent memory access. Modern GPUs also have parallel pipelines that allow them to achieve a high degree of data parallelism on linear control flow, where linear control flow consists only of data-independent control structures. For example only for loops with fixed iteration count. However, SIMD machines perform rather poorly on data-dependent control flow since data-dependent control flow is usually handled in a SIMD machine by idling processors and other resources.

GPUs have recently incorporated more programmable capabilities. The latest gen-

eration of GPUs have powerful programmable shading units with floating-point support. Even though these features are targeted primarily at transformations and surface shading, they are powerful enough to make the GPU an interesting platform for general purpose programming [26]. The list of applications that have been “mapped” onto GPUs is large, and includes linear and non-linear equation solvers [10], sparse matrix solvers [2], ray tracer implementations [21], and global illumination solvers [4]. In addition to their programmable capabilities, GPUs have hardware support for various operations such as look-up tables, texture filtering, etc., that might be useful assets in general purpose programs. For example, texture look-up provides low cost linear, bi-linear and tri-linear interpolation.

A SIMD GPU is, therefore, powerful parallel processors with great computational potential, but it suffers from the shortcomings of most data-parallel systems: poor performance on data-dependent control flow because of resource idling. We need a way of truly avoiding unnecessary computations.

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 multi-pass streaming computation framework, the control graph can be interpreted as a data flow chart: nodes are independent pure SIMD programs called *kernels* that operate on large sets of data. This graph is called a *streaming graph*.

The contribution of this thesis is a design for a system that transparently transforms the GPU ¹ into a general purpose stream processor with asymptotically efficient data-dependent control structures.

An implementation of this system was built on top of the Sh library. Sh [15] is a high-

¹Not all of the GPU resources are used in this framework.

```
ShProgram ifp = SH_BEGIN_PROGRAM("gpu:stream"){
  ShInputTexCoord3f input; // input
  ShOutputColor3f ocolor; // output

  // Kernel A: some initial computations

  // ... computations ...

  // Split the data flow into two branches
  SH_IF( ... condition... ){

    // Kernel B
    // ... computations ...

  } SH_ELSE {

    // Kernel C
    // ... computations ...

  } SH_ENDIF;

  // Kernel D
  // ... Final computations ...

} SH_END_PROGRAM;
```

Listing 1.1: Sample Sh code

level GPU programming language implemented as a C++ library. The Sh syntax supports data-dependent conditional and iteration statements. Sh also provides an implementation of a stream/shader algebra that exposes an API to modify programs [13] and apply programs to streams.

A general purpose Sh program is represented as a streaming graph that is executed in multiple passes. Listing 1.1 shows a simple Sh program that corresponds to the data flow

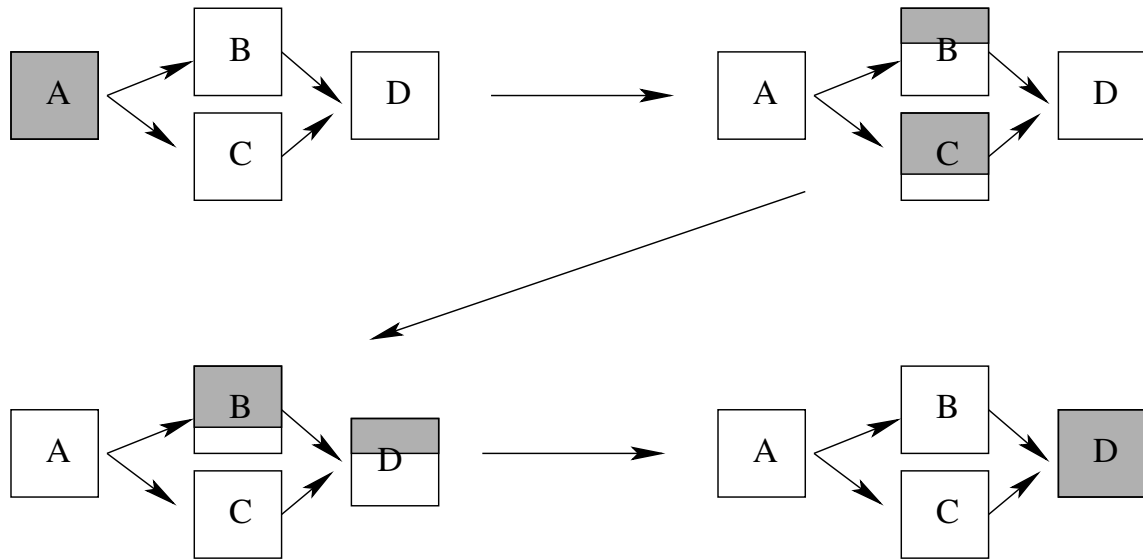


Figure 1.1: Data flow of a simple streaming graph with conditional control.

in figure 1.1. First, kernel *A* process all the input data and splits the data stream based on a data-dependent condition. Then, one of the kernels *B* and *C* is scheduled to process its data. In our example, kernel *C* is the next running kernel. After that, kernel *B* process its data and than *D*. Each instance of a kernel execution is called a pass and it is possible to execute any program with arbitrary data-dependent control structure in multiple passes. At each pass a sub-stream of the original input stream takes one step through the graph. This process is repeated until all intermediate buffers are empty and all input data is consumed.

This thesis deals with three aspects of this process:

Stream Graph Construction. The program output by the Sh compiler has to be transformed into a canonical form that is understood by the scheduler. This involves partitioning the program and altering its structure.

Kernel Scheduling. In each pass, a kernel will be selected by the scheduler based on heuristics to maximize throughput.

Resource Maximization I used and extended the conditional streams concept [9] to compactly pack data into different buffers to handle conditional outputs and eliminate idling.

The Sh compiler, shader and stream algebra and scheduler together form a complete programming system for GPUs that supports imperative paradigms including data-dependent control flow structures. My task is to find a mapping of the data-dependent constructs to current SIMD GPUs.

In chapter 2 I describe more formally the problem of data parallelism and various approaches to expressing and implementing parallel computations. I also survey the basic architecture of contemporary GPUs (circa 2003/2004) (Section 2.2) and I present an overview of Sh (section 2.5), the high level programming system that I used to target real GPUs.

There are two implementations of our systems. One runs on a software simulator (Sm) that I used initially to develop the scheduling algorithm. Another implementation runs on top of Sh and targets real GPUs. In chapter 3 I describe in detail the Sm implementation and in chapter 4 I describe the system that targets real GPUs. In chapter 5 I advance conclusions and outline future work.

Chapter 2

Background

2.1 Parallel Architectures

One way to increase the performance of a given system is to distribute the workload across multiple computational units. Several computers working cooperatively to solve a problem or a multi-threaded program on a multi-processor machine are examples of such parallel computing.

There are several models of parallel computing: a collection of independent machines connected by a network, one machine with one memory space but several processors, or one machine with one memory space and one processor, but several parallel functional units for performing arithmetic. GPUs are a combination having one memory space and multiple processing units, but also each processing unit has multiple functional units that can operate in parallel. However, GPUs have restrictions on their memory model. For instance, they cannot read and write to the same memory locations simultaneously, but

must do so in separate “passes”.

There are many architectural classes of parallel systems. A simplified classification due to Flynn [24] is based on the number of control units and number of processing units available:

- Single instruction stream (abbreviated as SI)
- Multiple instruction streams (abbreviated as MI)
- Single data stream (abbreviated as SD)
- Multiple data streams (abbreviated as MD)

These criteria yield four classes: SISD, SIMD, MISD, MIMD. This thesis focuses on Single Instruction Stream Multiple Data Streams (SIMD) and specifically, stream architectures, which are a specialized form of SIMD architecture that extends the SIMD model with *sequential* memory access.

2.1.1 Single Instruction Stream Multiple Data Streams (SIMD)

The classical SIMD architecture [24] is composed of an array of processing elements that execute the same instructions simultaneously. The processing elements are programmable and have local memory. This array of processing elements are a co-processor to a host computer that generates the instructions and to a memory hierarchy for data I/O. The connection to the memory hierarchy is generally done using a wide data path to allow parallel data fetching by the processing units. An extension of the traditional SIMD architectures is SIMD Within a Register (SWAR) [6]. SWAR adds an additional layer of parallelism:

each processing unit has parallel pipelines allowing parallel vector computations on the input data. The data path from memory to the processing units is subdivided once to feed each individual processing unit and it is subdivided further into fields that can be processed in parallel. This design is useful for graphics and multimedia computations where 3D or 4D vectors are often used as primitive types. Even though data independent control flow SIMD architectures are considered to perform efficiently, e.g., high resource utilization, they tend to perform poorly on data dependent control flow. On a SIMD machine, all processing elements execute the same instructions, therefore processing is more efficient if the input stream is made of homogeneous records: records that have the same control path within the control graph of the program. A branching instruction, for example, can break the homogeneity of the input data and some of the processing units have to idle, waiting for the data to be synchronized. For example, in fig. 2.1, the input is a stream of integers. Since, in this example, only the even integers are incremented and the stream is made out of a mix of even and odd integers, some processing elements will idle.

	i	404	703	153	234	789	550				
<code>i = i % 100;</code>	4		3		53		34		89		504
<code>if(i % 2 == 0){</code>	5		Idle		Idle		35		Idle		51
<code> i+=1;</code>											
<code>}</code>	10		6		106		70		178		101
<code>i*=2;</code>											
Processing Elements	0		1		2		3		4		5

Figure 2.1: SIMD execution of a simple program that uses conditionals

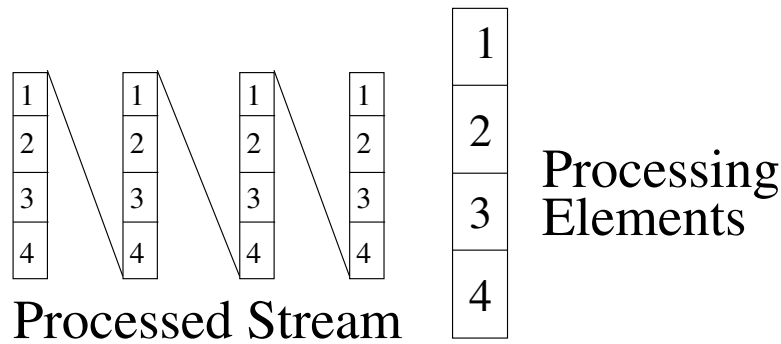


Figure 2.2: The records of a stream are processed in blocks.

2.1.2 Stream Processing Model

A programming model that maps well onto SIMD architectures is the Stream Processing Model. A stream is a sequence of elements of the same type. Each element, however, may have a number of subfields. Programs operating on streams are called kernels [18]. Kernels are small programs that are repeated for each element of its input stream to produce an output stream. The stream records are not processed sequentially, they are processed in blocks as illustrated in figure 2.2. This is called *strip mining*. This computational model achieves high parallelism and throughput by maintaining a high data bandwidth and allowing deterministic pre-fetching and block oriented transfer. One of the recent experimental processors designed to use the streaming computational model is the Imagine processor [23, 18] developed by the Stanford Computer Systems Laboratory. Imagine is a single chip processor that has multiple arithmetic logic units (ALUs) connected to a *Stream Register File* (SRF). The SRF is a block of memory that acts as a managed cache with high bandwidth to the arithmetic units. The SRF is partitioned into *stream buffers* allowing each of the arithmetic units (also called *arithmetic clusters*) to fetch stream elements from the SRF

and to then execute a kernel on an element of the stream. In this model, no cross-feeding of the arithmetic clusters is allowed: the ALUs can fetch data only from their own stream buffer.

Their experiments showed a significant increase in performance for algorithms that fit the stream processing framework. An OpenGL implementation was created on the Imagine stream processor [19] and the analysis showed that the only stage of the pipeline that did not benefit from the streaming computation model was the rasterizer. This deficiency may lead to the idea that a separate hardware rasterizer on the chip might be useful, but for other graphics tasks the streaming computational model appears to be quite appropriate.

The significant potential increase in performance makes the streaming computational model attractive. The performance of such a stream-based computer system relies heavily on data locality. Fortunately, most multimedia and graphics applications are characterized by little data reuse and high computation to memory access ratio, making them ideal for a stream computation framework. However, not all algorithms fall into this paradigm and particularly data-dependent branches and random memory access often break data locality. On average, in typical general purpose programs, one in five instructions is a branching instruction, so it can be seen that the scope of the naive stream processing model is relatively narrow.

Some work has been done to address this problem. An interesting alternative approach for implementing conditional execution efficiently is the *conditional stream* [9]. Conditional streams introduce a few incremental changes to the streaming model described before. From the perspective of the streaming processing model, branching instructions can induce an implicit partitioning of the stream into two or more sub-streams. Processing

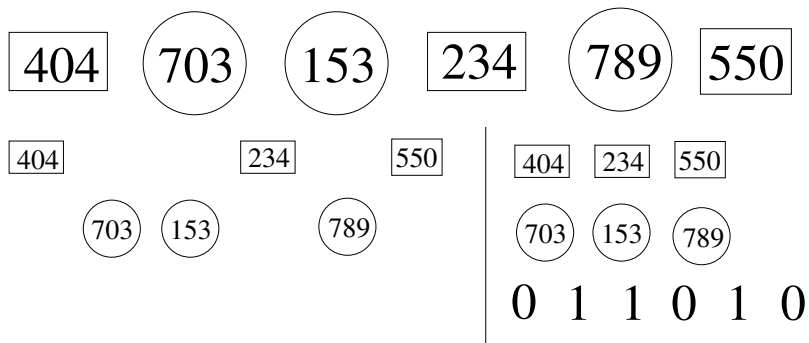


Figure 2.3: Stream Partitioning and Compression

the main stream in a SIMD fashion results in resource idling (Fig. 2.1). However, if these interleaved sub-streams could be extracted and compressed into smaller but homogeneous streams, the streams can be processed with maximum resource utilization. For example, figure 2.3 shows how the idling in the previous example (fig 2.1) can be avoided. Again, we are given a the same stream of integers subject to the same even/odd partitioning. The top row shows the original stream. The original stream can be separated into two groups as shown in the bottom left part of the figure. This will result in two sparse streams. The resulting two sparse streams can be compacted resulting in two dense and homogeneous streams that can be processed optimally. However, after processing, the two sub-streams have to be merged back together to restore the structure of the original stream. To achieve that, we encode the merging order into a binary stream as illustrated in the bottom right corner of the figure.

Conditional streams are basically streams that can be accessed based on a condition local to the stream elements. This conditional access allows stream compression and expansion via two operations denoted as *switching* and *combining*.

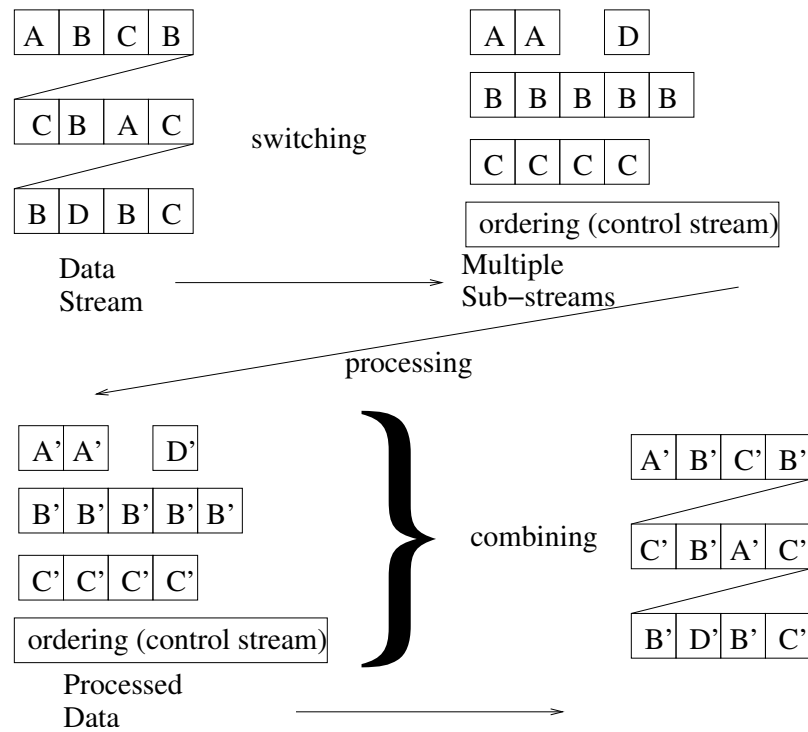


Figure 2.4: Switching and Combining Conditional Streams

Switching: This operation uses conditional output to compress a stream. Output data is routed into one or more destination streams so that each stream consists of homogeneous data.

Combine: This operation uses conditional input to merge two streams. When a stream is partitioned and compressed, the order of the original stream is lost (Fig 2.3). The order must be encoded in an additional third *control stream* that contains the interleaving order of the compressed streams. This control stream usually requires a small number of bits to encode.

Figure 2.4 shows how the switching and combining operations integrate with the tradi-

tional stream processing

An implementation of conditional streams requires some incremental hardware changes to the traditional SIMD architecture. Kapasi [9] argues that the changes are small and they focus mainly on data cross-feeding of the processing elements. I rely on his analysis to make a similar claim for GPUs.

In my opinion, the terminology that Kapasi uses when referring to stream compaction is ambiguous. He uses the terms compression and expansion. The term “compression”, especially, can be easily misinterpreted. Therefore, I propose a simpler terminology: packing and unpacking. Packing is the spatial compaction of stream data in order to avoid redundant computations on null records. Unpacking is the opposite operation that merges two previously compacted streams with a common control stream into one stream.

2.2 GPU Architectures

GPUs are co-processors designed and optimized to process 2D and 3D geometry. Data is transferred between the main memory and the GPU memory via a dedicated Accelerated Graphics Port (AGP). On current GPUs, data transfer between the CPU memory and the GPU is much slower compared to the processing speeds of either the CPU or the GPU. This is an important limitation that shows up in a few places in this thesis.

The overall functional organization of modern GPUs as a pipeline has not changed much in recent years. However, the performance and feature set supported by the hardware improves dramatically with every generation. Modern GPUs are programmable and they employ parallel pipelines for better performance and a SWAR model of execution.

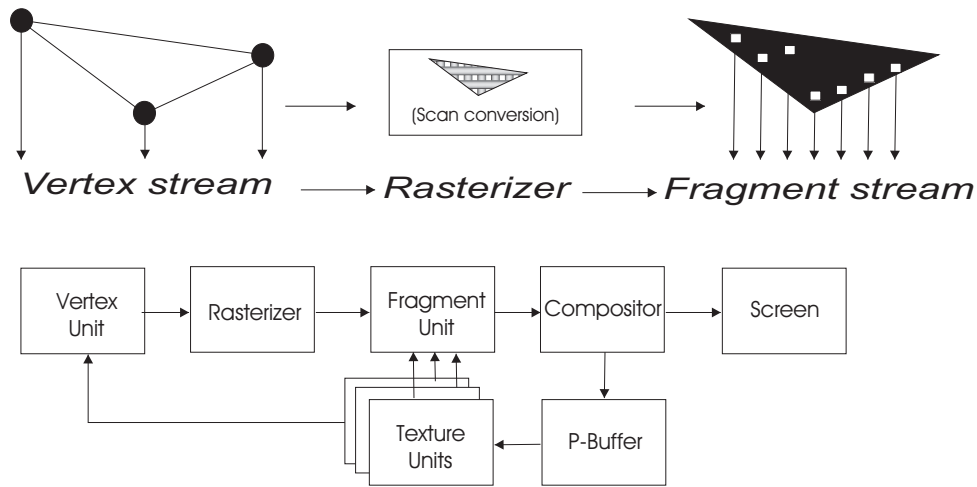


Figure 2.5: Simplified GPU pipeline. Top: generation of fragment streams. Bottom: Schematic data flow of a GPU.

Conceptually, a graphics pipeline works as follows (Figure 2.5): the user discreetly specifies the geometry as a set of points called vertices and a set of faces. Faces are indexed values in the array of vertices. The vertex stream specified by the user is processed by the vertex units using a vertex program or shader ¹ specified by the user (including the application of the model-view, projection and viewport transformations). The vertices are grouped together in triangles and are scan-converted by the rasterizer (the clipping stage is omitted for simplicity). The outcome of this operation is a large stream of elements called fragments, which represent individual pixels on the screen. These fragments are further processed by the fragment units using a fragment shader specified by the user. After that, the fragments are written into a memory buffer, also called a frame buffer, that is copied

¹In the computer graphics literature, GPU programs are commonly refer to as shaders.

into the video memory producing an image on the screen. I refer to this process as one rendering pass. Some algorithms cannot be implemented in one rendering pass, therefore, multi-pass rendering techniques have been developed. The traditional multi-pass rendering technique uses a compositor module located on the GPU to combine the fragment of the current pass with the fragments already in the frame buffer from previous passes. A more powerful technique is to take the data from the frame buffers and feed it back to the fragment or even vertex units. However, the frame buffers are integer buffers with (usually) 8-bit precision per component. For visualization, this precision is sufficient, but for general purpose computations or for fragment feed-back this precision is insufficient. However, modern GPUs have a special type of buffer, called a P-Buffer, that can be used if higher precision is required. P-Buffers can be used as render targets and support high precision floating-point data. P-Buffers can later be used as floating point textures facilitating multi-pass rendering. However, P-Buffers cannot be used to directly drive a display.

The above description of the GPU pipeline is greatly simplified. Several features like occlusion culling and stencil tests have been omitted for clarity. The most relevant parts of the graphics pipeline with respect to my research are the vertex and the fragment units since they are the programmable processors on the GPU. The vertex and fragment units have similar programming capabilities. In addition to their mutual programming features, the fragment unit can perform texture look-up operations², but has the disadvantage that it cannot change the position of the fragment on the screen. This restriction is an important limitation because it means discarding fragments from a fragment stream does not remove their storage, resulting in poor data locality.

²The recently released GeForce 6800 can also do texture lookup in the vertex unit, which is part of a general trend of convergence in the abilities of the vertex and fragment units.

Both vertex and fragment shader units can be programmed directly using an assembly programming interface. Instructions can have one, two or three source registers and up to one destination register. The primitive register type is a four-component vector of floats. The instruction set is a mix of scalar and vector operations. For example, the ADD instruction adds two vectors component-wise while a dot product operation takes two vectors and returns a scalar. Vector operations are performed in parallel on all four components. Most current GPUs efficiently support swizzling and write masking. Swizzling allows an arbitrary reordering of the vector components and write masking restricts the write operation to an arbitrary subset of components. For example, if $v1$, $v2$ and $v3$ are 3 component vectors, the instruction $MUL(v1.xy, v2.xx, v3.zw)$ does $v1.x = v2.x * v3.z$ and $v1.y = v2.x * v3.w$ while $v1.z$ and $v1.w$ remain unchanged. These features provide a great deal of flexibility to the programming interface.

This style of vector computation is very useful in multimedia applications and something similar has been supported on Intel and AMD CPUs as extensions to the traditional x86 instruction set. For example, the MMX instruction set [20] supports a SWAR style of execution on integer vector elements. The system delivers up to twice the performance of the traditional scalar execution model if the problem is appropriate for this vector approach. Intel extended the vector support to floats in their Streaming SIMD Extensions (SSE, SSE2) [22]. 3DNow! technology from AMD [17] is also an extension to MMX instruction set targeted to graphical applications that supports single precision floats. However, neither of the SSE nor 3DNow! extensions support efficient swizzling.

The SWAR/SIMD stream computational model raises a number of challenges in the areas of language semantics, data packing, compiler optimization, and scheduling. Among

these issues, this thesis deals with data packing and scheduling.

There are currently two standard APIs available to access GPU resources: OpenGL and Direct3D. Even though some important features are better exposed in Direct3D, due to wider support and portability our systems use OpenGL. These APIs expose a low level of abstraction.

2.3 GPUs and Stream Processing

The nature of the data that GPUs operate on make them suitable for a streaming computational model [18]. Traditional vertex and fragment operations exhibit the sequential data locality characteristic of the streaming computational model, and the vertex and fragment processing pipeline can be naturally broken down into kernels [23], which makes them suitable for a stream processing architecture.

In terms of the streaming computational model, data inside the GPU can be classified as vertex streams and fragment streams. The vertex streams are processed by the vertex units and the fragment streams by the fragment units. In a typical graphics application, the fragment streams can be much larger than the vertex streams, therefore, GPUs typically have more fragment units than vertex units.

Even though the implementation details of current GPUs are generally kept confidential, performance analysis and published documentation suggest that most GPUs are indeed implemented roughly following the streaming paradigm. Therefore, I tried to use the fragment shading unit, in particular, as a streaming processor. I also want to implement an efficient and transparent programming framework that supports the entire palette

of imperative control constructs including data-dependent conditionals and iterations.

Arguably, the vertex unit and the rasterizer can potentially be used as powerful players in such a framework. For instance, the vertex unit can be used as a scattering engine: vertices coming into the vertex unit can be placed anywhere on the screen. However, for our initial attempt I restricted my prototype to a simpler framework involving only the fragment unit.

The GPU streams abstraction implemented in Sh uses GPU fragments as stream elements. Streams are stored in textures for reading operations and in P-buffers for writing operations. The branching instructions are handled using conditional assignment and multiple output buffers. Conditional assignment allows dynamic selection of an assignment based on a data-dependent condition, and multiple output buffers allow multiple output targets. Each output variable has an output buffer to write the result to. In this way the data flow is routed to the appropriate kernels. For example, in fig. 2.6 data from A is conditionally routed based on a run-time boolean variable c . However, the two generated sub-stream are not compact. Subsequent processing of these sparse streams results in redundant computation since the null records cannot be avoided.

While current GPUs support data-dependent texture look-ups that allow random access on reads, the positions of the fragments on the viewport cannot be changed. Therefore,

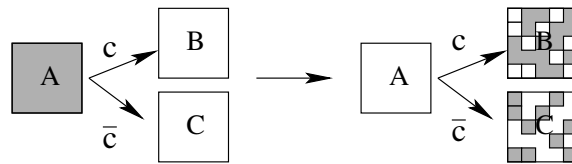


Figure 2.6: Conditional Flow

kernels implemented using fragment shaders can modify only the current record that is processed. Replicating the behavior of conditional streams on GPUs requires hardware support for packing and unpacking streams. Unfortunately, the structure of the fragment stream generated by the rasterizer is fixed, therefore packing fragment streams cannot be done in one pass on current GPUs. This thesis follows a main thread where I assume that GPUs can be augmented with special hardware to support packing efficiently, and a tangential thread where I discuss possible strategies of packing on the GPU. For the main thread, I argue in section 4.1.1 that the hardware modifications to accomplish efficient packing are small.

Most GPUs also have a limited number of resources that can be used in a single pass: instructions, input/output registers, texture look-ups, and texture units. Therefore, even if a program does not use data-dependent control flow, it may not be able to run due to resource exhaustion. Chan et al. [5] showed that it is possible to take an arbitrary program with linear control flow and one output, and partition it efficiently and nearly optimally into executable blocks and run it in multiple passes. My system can be extended to incorporate such *virtualization*. However, to limit the scope of this thesis, I assume that the system already has a virtualizer and can support kernels of arbitrary length. In practice, in my prototype, Sh will detect if the resource limits of a kernel are exceeded and it will exit with an error.

2.4 Parallel Performance

The performance of a parallel architecture is governed by Amdahl's law [1]. A given program can be partitioned into a parallel part that can be independently distributed over several computational units and a serial part that is not parallelizable. For example, consider a problem where the input is a set of points that require some processing and the output is the distance between the two closest points. The processing of the individual points can be done in parallel, but finding the two closest points is more difficult to parallelize. Amdahl's law states that

$$S = \frac{N}{(B * N) + (1 - B)} \quad (2.1)$$

where S is the speed-up if using N processors as opposed to just one, and B is the serial fraction of the program; therefore $0 \leq B \leq 1$. ' If the program is fully parallelizable, the speed-up factor is N and if the program is fully serial, the speed-up factor is 1. Also as N goes to infinity, the running time of the parallel part converges asymptotically to zero and the serial part does not change; therefore, the lower bound of the running time is the serial part of the program. The execution model of a parallel program is an alternating sequence of serial and parallel parts. According to Amdahl's formula, the main challenge in writing a parallel application is to minimize its serial part. The serial part of an application often includes the overhead associated with communication and operating system tasks. Generally speaking, access to shared resources forces serialized execution.

In my system, the scheduler is an example of a serial resource. While the scheduler is busy selecting a candidate from the kernel pool, the entire pipeline is waiting. This serialization can make the scheduler the bottleneck of the system. Since the scheduler

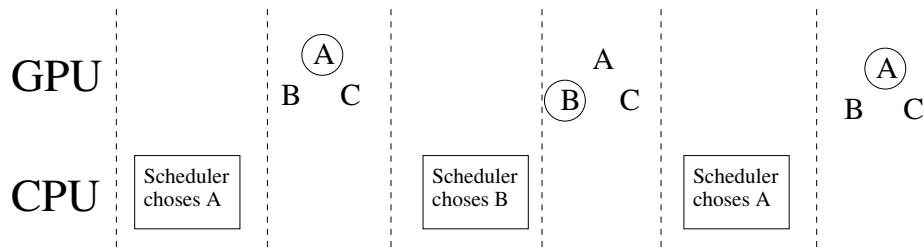


Figure 2.7: The scheduler and the GPU programs running serially

runs on the CPU and the data processing is done on the GPU they could run in parallel. Therefore, I can avoid scheduling serialization problems by requiring the schedule to have a decision ready by the time the GPU needs more data. For example, figure 2.7 shows a snapshot of a program execution involving three kernels: *A*, *B* and *C*. At the end of each rendering pass, the scheduler makes a decision based on the most recent information available. However, this wastes numerous cycles on both CPU and GPU. In figure 2.8, the same kernels are scheduled one step in advance. This approach may lead to a less optimal scheduling sequence, but it increases resource utilization.

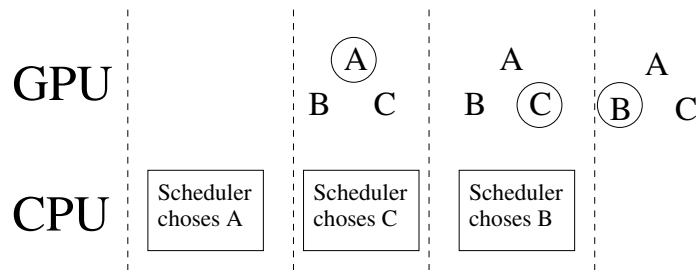


Figure 2.8: The scheduler and the GPU programs running in parallel

2.5 Sh System

Sh started as a prototype for a shading language [15] and has evolved into a complete GPU programming system with stream semantics and support for multi-pass execution. Sh targets various kinds of graphics hardware as well as the Sm simulator. I used Sh as the platform for my dynamic multi-pass scheduling algorithm targeting real GPUs. The following section emphasizes the critical features of Sh that facilitated the implementation of my algorithms. The complete reference manual of Sh [12] including its standard library can be found at: <http://libsh.sourceforge.net/>.

2.5.1 Stream Abstraction

Sh has a stream abstraction that converts data streams into fragment streams that are passed through the graphics card for processing. The data is transferred to the card as texture data and the computation is driven by rendering one or two rectangles into an OpenGL buffer. When compiled to a stream processing target, the program is modified so it fetches its input data from a texture. Internally, a stream is a collection of channels and a channel is an ordered sequence of a primitive type. Primitive types in Sh are tuples of up to four floating point data components. On GPUs, a stream of an arbitrary complex type (e.g. like a structure) is stored as several independent channels. This requirement is due to the memory organization of GPUs. However, the Sh stream abstraction hides all these low level details.

Sh provides a high level API to manipulate streams and programs. For example, if $S1$ is a stream of 3D vectors, $S2$ is a stream of scalar floating point data, and P is a program

that computes the Euclidean norm of a vector, the operation $S2 = P \ll S1$ makes $S2$ a stream of floating point values that corresponds to the norms of the vectors in $S1$. Also, one can create streams with more complex composite records by combining several other streams or combining a stream with one or more channels. For example, if a stream $S1$ is made out of 2 channels of 3-tuples and $S2$ is made out of 3 channels of 3-tuples and C is just one channel of 3-tuple than $S1\&S2\&C$ is a stream of six 3-tuples. The channels of a stream need not be of the same type. The number of channels in a stream is called the width of the stream. Therefore, an expression like $(S1\&S2\&S3) = P \ll (S4\&C1\&C2)$ is valid only if the semantic type and width of $S1$, $S2$ and $S3$ matches the output of the program P and when the semantic type and width of $S4$, $C1$ and $C2$ matches the input of the program P . If these conditions are not met, a run-time exception is raised. This error is not capture at compile time because Sh compiles the GPU programs at run-time.

2.5.2 Shader Algebra

The stream abstraction provides an API to combine streams with programs. The shader algebra operations [13] in Sh provide a standard API to combine programs. These operations are a powerful mechanism allowing the manipulation of programs as first class objects and provide the programmer with tools to combine programs in a flexible way. These operations are useful since the programs output by the Sh compiler are eventually subject to a series of complex transformations that split it into several pieces to be executed by the multi-pass scheduler.

The shader algebra defines two operators: *connect* and *combine*. The operators are binary and output objects of the same type as their inputs: Sh programs. Connect is

defined as functional composition: the outputs of the first program are fed into the inputs of the second program. The output signature of the first program must match the input signature of the second program. If they do not match, the system raises an exception. Combine is defined as a “union” operator: given two shader programs A and B , the result program C contains the statements of both programs and the list of inputs and outputs is the concatenation of the list of inputs and respectively outputs of the two programs. The connect operator is denoted by \ll with inputs to the left and outputs to the right: $a \ll b$. The combine operator is denoted by $\&$: $a\&b$.

These two operators have rigid semantics. For example, the connect operator requires the input and output of the two programs to match. Often, a user may need to perform little adjustments to permute the outputs or ignore some output values. Another situation is illustrated in Figure 2.9. Connecting A and B requires ignoring an input. Connecting B and C requires producing a new input, maybe a duplicate of one of B 's outputs or a constant value. Connecting all three programs could require that A 's third input to be carried forward to C . These operations can be expressed by defining small glue programs, but are so common that Sh provides a set of manipulator objects that build them automatically. Example of manipulators are:

shDrop: Discards a channel and rearranges the rest to close the gap

shKeep: Copies channels from the program's input to its output

shSwizzle: Rearranges the order of inputs and outputs.

For the interested reader, further stream algebra details can be found in the *Shader Algebra* paper [13]

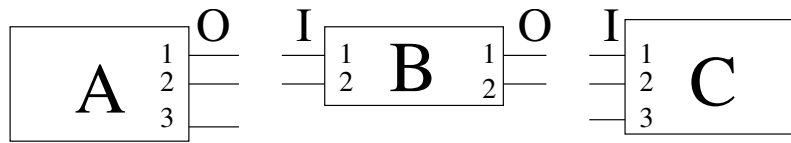


Figure 2.9: $B \ll A$ matches $oA1$ and $oA2$ with $iB1$ and $iB2$ and the resulting shader has a third output equal to $oA3$. $C \ll B$ is invalid since the third input of C is not bound to any variable. $C \ll (B \ll A)$ matches $oA3$ to $iC3$.

2.6 Organizational Notes

There are two implementations of the basic ideas discussed in this thesis. One runs on a software simulator, Sm , and the other runs on top of the Sh system and targets real hardware. The motivation to build a software simulator is twofold. On the one hand, when I started this research, the state of the art in graphics hardware was missing some essential features and, on the other hand, building a software simulator allowed us to experiment with various possible supplementary hardware features, and to analyze which of these features would be worth having on a GPU. The Sm system does not convert the output of the compiler into streaming graphs. Instead, the streaming graphs are created “by hand”. Chapter 3 describes the Sm implementation of my system. This work represents an initial effort in designing the multi-pass scheduling system.

Before describing either of the two systems in detail, some notational conventions are necessary. The two systems use different types of containers for streams. The simulator uses a one dimensional memory buffer as a stream container and it fills it in a FIFO fashion with data records. The GPU prototype uses textures and frame buffers as containers for streams. Both textures and frame-buffers are two dimensional containers that are ordered left to right and top to bottom (scan line order). In the GPU implementation, records

are distributed over multiple channels that in turn correspond to individual textures and respectively frame buffers. The difference between the two is illustrated in figure 2.10.

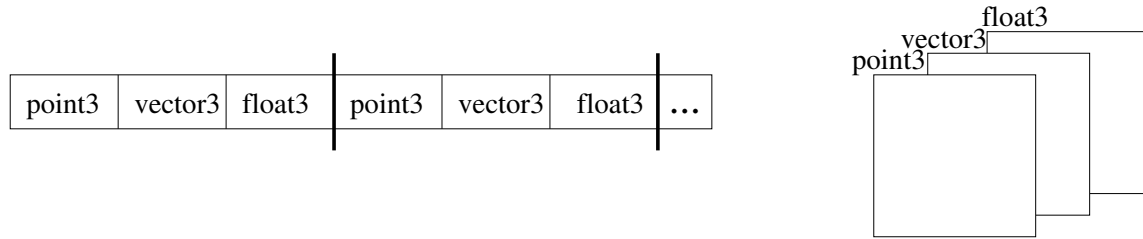


Figure 2.10: Left: serial data stream; Right: sliced data stream.

On the left, there is a serial data stream that stores the records clustered together. Accessing the N th record requires an offset of $N * sizeof(\text{sequences of records})$

This yields two different notations as illustrated in figure 2.11. On the simulator, a kernel and its respective input buffer is denoted by a circle and a rectangle. On our GPU prototype, the kernel and its associated input buffer is represented by a square.

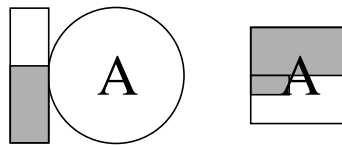


Figure 2.11: Left: kernel and buffers in Sm. Right: kernels and their respective input buffers are represented by a square on the GPU prototype.

Chapter 3

The Sm System

3.1 The Sm Simulator

Sm, the stream processor software simulator that I used, implements a packet-based stream architecture conceptually similar to Chromium [7]. The system consists of a number of modules that communicate by means of self-identifying variable-length packets. For my tests I used a configuration of modules functionally equivalent to a GPU: vertex unit, rasterizer, fragment unit and compositor. At a high level, Sm is organized as a C++ library and it exposes an OpenGL-like API.

All communication in the system (including download of texture data and shader programs) is by means of these packets; each API call simply generates an appropriate packet or sequence of packets, and modules forward packets they do not understand to the next module. The last module has a connection back to the API for the purpose of uploading data.

A given configuration of the graphics pipeline, including rasterizer state, vertex and fragment programs, and compositing operations, corresponds to exactly one kernel in the streaming graph. Given a streaming graph, a dynamic scheduler runs the kernels sequentially until all input data is processed. The order of the kernels is decided at run-time and it uses a greedy heuristic to maximize throughput. See section 3.2 for details.

A single kernel can read one stream but it can write to at least two output streams and it can output more than one record to each output stream. Permitting a kernel to conditionally write larger records to an output than are read on an input allows data amplification. In this context, the data amplification factor is defined as the ratio between the sizes of the output and input data. The size of this data is expressed in words rather than records. Data amplification together with feedback paths can be used to implement some recursive algorithms such as the adaptive tessellation algorithm described in section 3.3.

Data streams are stored in off-chip memory buffers called stream buffers. It is desirable to support a limited number of fixed size on-chip stream buffers to improve effective bandwidth. I call these on-chip buffers *stream registers*. The number and size of stream registers is hardware-dependent. Access to stream registers can be significantly more efficient than access to off-chip memory, but the user needs to be careful to avoid buffer overflow, since stream registers are of fixed size. To avoid processing partial records, the stream registers only store complete records.

Allowing data amplification factors greater than one, the amount of data that needs to be stored can grow without bound. This problem can be addressed using a dynamic memory allocation scheme (for instance, growing stream buffer sizes as needed). However, the queue (FIFO) semantics of stream buffers are not ideal for recursion. FIFO storage leads

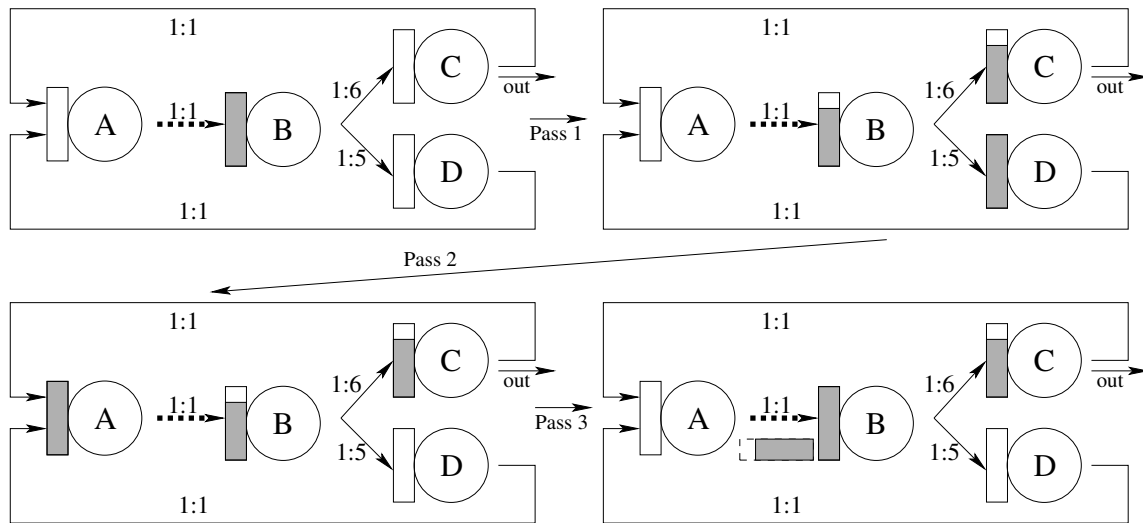


Figure 3.1: An example of multi-pass execution with data amplification.

to a breadth-first exploration of a recursion tree, which consumes memory exponential in the recursion depth. The more usual depth-first recursion requires only linear storage with respect to the recursion depth. However, a simple LIFO (stack) buffers cannot be used instead of FIFO buffers because to achieve high performance we need to process a large number of records in one pass to achieve high throughput. Depth-first recursion requires to immediately use an output as an input to another invocation of a kernel, a strong data dependency that leads to difficulty scheduling parallel computations. To deal with these issues, Sm uses a *stack* of stream registers. The scheduler will allocate a new stream register on the stack as needed for kernels writing to one of the arcs (selected by the user) of the feedback cycle. This approach leads to a hybrid depth and breath order for exploration of the recursion tree.

It is important to emphasize that this stack based allocation scheme is required only in the case of data amplification combined with feedback. Otherwise, a simpler list based

allocation scheme is more appropriate. Additional work is required to integrate the two schemes into the same algorithm.

My approach is to classify the arcs of a streaming graph into two categories: regular arcs and stack arcs. In a streaming graph, arcs correspond to buffers that hold input and respectively output data of their adjacent kernels. Arcs pointing to the same node share the same stream register. Regular arcs correspond to a statically allocated stream register while stack arcs have a stack register allocation protocol. The active register on top of the stack is the only one valid for accessing. The system spills stream registers deeper on the stack to off-chip stream buffers when necessary. In this way, the depth of the stack can be virtualized and the programmer can treat it as being infinite, given sufficient off-chip memory for spilling.

Figure 3.1 illustrates this process with a simple example. The streaming graph has four kernels with their associated data amplification factors as illustrated in the figure. The arc AB is marked as a stack arc. All other arcs are regular arcs. The figure shows a snapshot of three rendering passes among all the rendering passes required to run that program. After the second pass, no kernel can consume much data. Most buffers are almost full and even though the buffer at BD is empty, it is unknown if the data from B will be routed through C or D , and therefore, in the worst case there is little data throughput. Therefore, the stack is pushed and a new empty stream register will be added and therefore the kernel A can be scheduled with very high data throughput.

A detailed analysis of the scheduler is presented in section 3.4. For more information on the Sm architecture, the interested reader can consult [14].

The current trends in chip design seem to be converging towards a cell model [25]

where a large number of computational units with high communication bandwidth between them are placed on a single chip, perhaps with multiple such units connected by a point to point network. A strong indication that this is true is the joint project of three of the biggest players on the market: Sony, IBM, and Toshiba to build such an architecture (http://www-3.ibm.com/chips/news/2001/0312_sony-toshiba.html). It may be possible to map stream architectures onto this grid model as well. Similar to Sm, in this scenario the kernels will no longer be simple programs, but rather entire network states: multiple programs and data paths. Scheduling will be more difficult because of the additional degrees of freedom, but a simple greedy approach might work in this case as well.

3.2 Scheduling Strategies

The goal of the scheduler is to minimize state changes by processing the maximum number of records at each pass. I implemented a greedy approach where the kernel that has the highest probability to consume the most data is selected to execute.

I call the maximum amount of data that can be consumed by a kernel in a given pass as the weight of the kernel. The weight of a kernel is expressed in words. At every pass, the scheduler estimates in a conservative way the weight of each kernel and it schedules the heaviest one. The estimate is the minimum between the input buffer size and the free space of each of the output buffers multiplied by their amplification factors. This last number is rounded down modulo the size of the record of the respective kernel. I do this to avoid processing partial records.

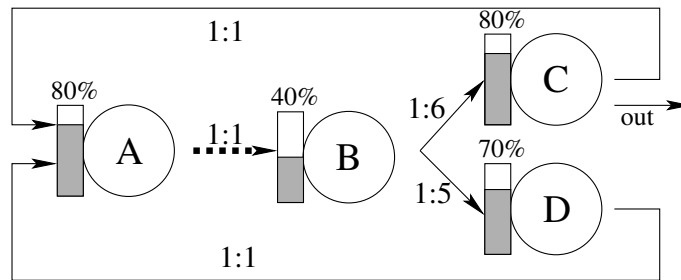


Figure 3.2: Greedy scheduling: in the worst case, kernel *A* can process the most data.

For each of the stack arcs, we have two scenarios corresponding to the cases where the stack is pushed or not. For example, figure 3.2 shows a streaming graph with four kernels and one stack arc. The numbers above the buffers represent how full the buffers are assuming that all buffers have the same capacity and ignoring for simplicity the issue of record boundaries. Looking at the figure, one can observe that if the stack is not pushed, in the worst case *A* can process 60%, *C* and *D* can process 20% and *B* can process roughly 3%. These estimates are conservative and assume the worst data path. In this case *A* is the best candidate with 60%. However, if the stack arc *AB* is pushed, *A* is still the best candidate, but it can output 80% rather than just 60%. In my implementation, for the stack arcs, new registers are allocated only when the top of stack register is more than 50% full. When the top buffer on the stack is empty, similar analysis is done in deciding to pop or not to pop the stack.

In the scheduler API, the fill fraction allowed is a parameter specified by the user. More details on scheduling strategy and performance analysis of the Sm system is available in [14].

3.3 Simulator Results

There are three interesting algorithms that I have implemented on the simulator:

Material mapping: Combines two shaders in a spatially variant way, dependent on a texture map. This example is a test of conditional control flow.

Julia Set: A Julia set evaluation is used to test iteration. This example is similar to the one implemented on real hardware.

Adaptive Tessellation: The triangles of a base surface are recursively split until the approximation (in screen space) to the displaced surface meets a given parallax error criterion. The algorithm that I have implemented can be found in [16]. This example illustrates the implementation of recursion, a feature that was implemented on the simulator, but I was not able to implement on a real GPU. It is implemented using data amplification.

The images generated by the test runs are shown in Figure 3.3. More details about these can be found in [14].

3.4 Performance Analysis

To generate meaningful numerical results, I assumed that actual hardware would have a 200MHz instruction issue rate, would use 16x SIMD execution for both vertex and fragment programs, would have a total off-chip memory bandwidth of 2GB/s, and would have a total bandwidth to on-chip stream registers of 20GB/s. These numbers are used by the

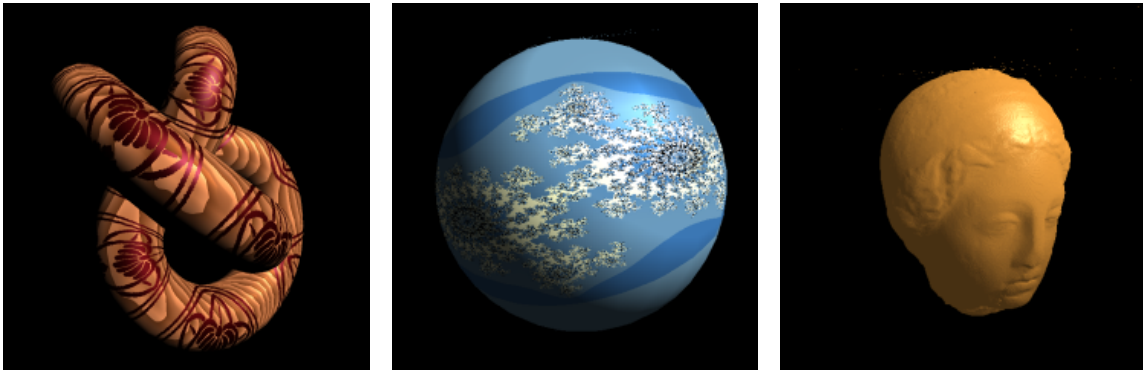


Figure 3.3: From left to right: material mapping, Julia texture, and adaptively tessellated displacement mapping. Renderings were antialiased using 2×2 supersampling.

scheduler heuristics. The clock rate and SIMD width assumptions are important for estimating kernel computation performance, while the off-chip bandwidth is important for estimating spill costs. Changing these numbers changes the passes that the scheduler chooses, since it is designed to adapt to different hardware configurations. With these assumptions, I wanted to evaluate the performance of the heuristically-driven scheduler in minimizing overhead and memory usage, and evaluate architectural tradeoffs, such as the number and size of stream registers.

Of the three algorithms tested, only the adaptive tessellation algorithm resulted in stream register spilling. I tested this algorithm under a number of different assumptions as to number of stream registers and the sizes of these stream registers, using the same view for each run. The results are shown in Table 3.1.

Using this test case, I can evaluate the scheduler's ability to minimize overhead in the form of costs to load new kernels and save and restore stream registers. Note that the total benefit — the total number of records processed by all kernels — is always the same for

each run. A large maximum recursion depth of nine and a small error threshold was used for this test run to stress the system. The results indicate that a large number of small registers decreases off-chip bandwidth requirements but requires more passes and kernel switches. In fact, for some of the register configurations given, spilling of registers is *required* since there are more arcs in the graph than registers. In general, the desirability of a given register configuration depends on the relative cost of register spilling and kernel switching, although there is a lower bound $2WR$ on stream register size given by the maximum record size R , the SIMD width W , and the 50% full rule for enabling stack pushes.

The relatively constant stream register utilization is a consequence because most of the available stream registers end up getting used for the stack. The stream register storage space utilization (60%) could be increased by raising the push threshold, although at the cost of greater scheduling difficulty. However, since only valid data is accessed from stream registers, low stream register utilization does not (directly) reduce stream register or off-chip memory bandwidth utilization.

I assumed a relatively high cost for kernel switching, to see how well our scheduler could minimize it. In practice, the total number of instructions required for this particular algorithm is low enough that a modestly-sized on-chip memory can hold them all. Recall that in a SIMD machine the instruction memory can be shared among all PEs.

Figure 3.4 shows the streaming graph for our test cases. The numbers on the arcs of these graphs correspond to the total bandwidth on each arc used when rendering the corresponding images in Figure 3.3. An explanation of the symbols used is provided in table 3.2

#Registers	Register Size (Kwords)	Total Memory (Kwords)	#Kernel Passes	#Kernel Switches	#Register Spills	Spill Bandwidth (MB)	% Register Utilization
8	4	32	9972	9222	981	6.86	59%
	8	64	4702	4427	531	7.11	59%
	16	128	2416	2212	243	7.00	59%
	32	256	1208	1122	121	6.86	58%
	64	512	618	568	58	6.57	56%
	128	1024	328	300	30	6.83	53%
16	2	32	22748	21174	1967	5.40	56%
	4	64	9972	9222	981	5.55	59%
	8	128	4876	4506	493	5.68	59%
	16	256	2416	2212	243	5.51	59%
	32	512	1208	1122	121	5.41	58%
	64	1024	618	568	58	4.62	56%
32	1	32	59643	52073	3685	2.90	50%
	2	64	22748	21174	1967	3.01	56%
	4	128	9972	9222	981	2.98	59%
	8	256	4876	4506	493	3.06	59%
	16	512	2416	2212	243	2.84	59%
	32	1024	1208	1122	121	2.54	58%

Table 3.1:

Performance of scheduler on adaptive tessellation test case relative to number and size of stream registers.

The top example shows the material mapping example. The *RastSplit* kernel routes the fragments based on a mask stored in a texture map. Kernels *HF* and *Wood* are shader programs that simulate garnet red using homomorphic factorization and wood respectively.

The middle example shows the Julia set example. The fragments are looped back until






Symbol	Meaning
	<i>Stream Buffer.</i> Contains an ordered sequence of data elements.
	<i>Array Buffer.</i> Random-access storage; pbuffer.
	<i>Computational Kernel.</i> Data-parallel program evaluation; single pipeline configuration.
	<i>Merge.</i> Combines two streams with the same record type. Does not preserve order of records.
	<i>Priority Merge.</i> Combines two streams with the same record type; does not preserve order. Preferentially reads from lower higher-priority input first.

Table 3.2: Symbols used in control graph diagrams.

either the maximum number of iterations is met or the fragment escaped the circle of radius two. Two versions of this algorithm were implemented. The first uses a feedback arc as shown in the figure. However, we also implemented in-place iteration, in which a special instruction indicates whether the shader should be repeated. If a true value is passed to this instruction, a new input record is not read, instead the shader is run again on the next

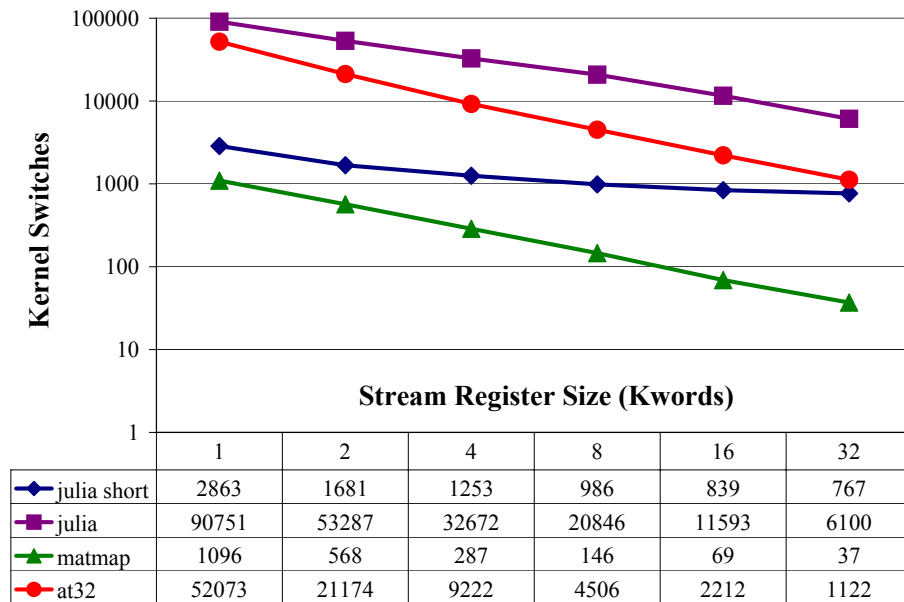


Table 3.3: Number of kernel switches vs. test case and stream register size.

SIMD cycle (without clearing the temporary registers first).

The bottom example shows the adaptive tessellation example. In this case, the input stream is a stream of triangles containing the three vertices are stream records. The *Oracle* shader decides whether the triangles need to be split. Based on the oracle's decision, the *Split* kernel routes the triangles to the appropriate kernel. Kernels *Tess2*, *Tess3* and *Tess4* split the triangles recursively as described in [16]. The *Bump* kernel tessellate the triangles and render them using bump-mapping.

Table 3.3 compares four test cases: the two versions of the Julia set implementation, material mapping, and adaptive tessellation. For the adaptive tessellation case, data for the case of 32 registers was used. For all other cases, 8 registers sufficed (without spilling). For the Julia set example, use of a feedback loop is compared with iteration in-place.

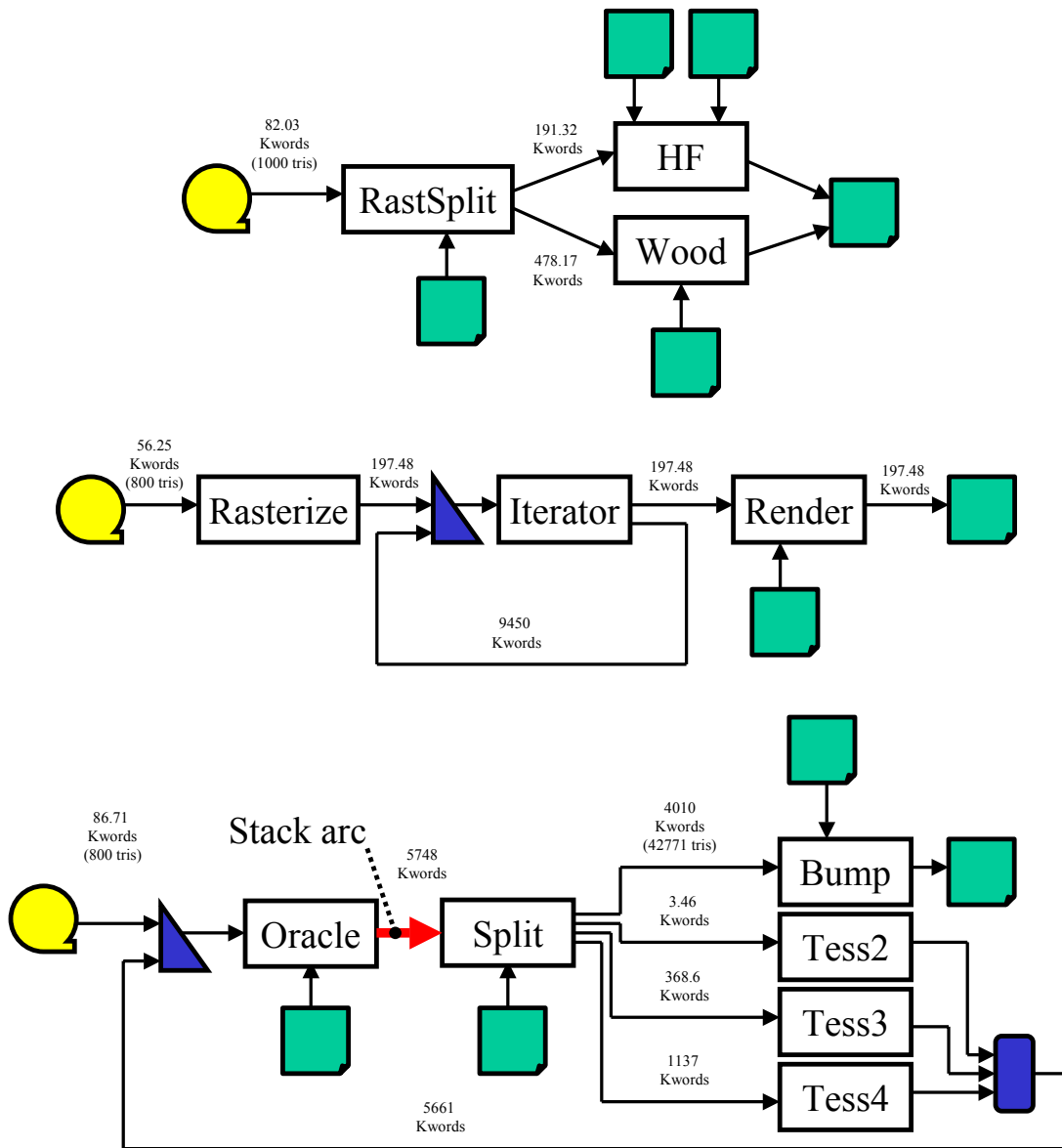


Figure 3.4: Streaming graphs for test programs. From top to bottom: material mapping, Julia set procedural texture, and adaptive tessellation.

In summary, the hardware specifications that we used to assess the performance of the system are not based on a real hardware design so it is difficult to assess their validity. I discovered that there is a strong tradeoff between the amount of memory available and the frequency of kernel switching and in-place iteration greatly reduces the number of switches. However, for nested iterations or for iteration bodies that include conditionals, use of the feedback loop approach is required.

Chapter 4

GPU System

This chapter describes the GPU implementation of my system implemented inside Sh, taking advantage of its features as presented in section 2.5: stream abstractions, program abstractions, and shader algebra operations.

The scheduling process has a pre-computation stage, where the control graph output by the compiler is transformed into a streaming graph, and a run-time stage, where a run-time scheduler searches for an optimal sequence in which to run the kernels.

Conditional execution can be handled in the streaming computational model using conditional assignment and multiple output buffers (section 2.3). However, the result of a branch is a set of sparse streams. Any further computations on sparse streams results in resource idling and thus loss of performance. To avoid that, a packing operation should be performed where the data of a sparse stream is compacted and the streams are reduced to smaller, denser streams.

4.1 Stream Packing and Unpacking

This thesis deals with two different scenarios that are treated separately. In the first scenario I assume that GPUs support “packing on write”: the null records are eliminated from the streams at write time. In this case, the overhead associated with the packing is negligible. I describe this scenario in section 4.1.1. However, this requires special hardware and cannot be implemented on current GPUs. In the second scenario I explore various methods to pack the data using current GPU features. This case is discussed only briefly in section 4.1.2.

4.1.1 Stream Packing using Beneš Networks

This section describes a method that can be used to modify current GPU architectures to support “packing on write”. Before elaborating on this approach, some preliminary information is presented.

In a multi-processor system, the processors are connected to memory via a network [8]. There are many designs of such networks, based on various characteristics such as network topology, timing protocol, switching method and control strategy [8]. We focus on one particular type of network to apply to the GPUs: The Beneš network (Figure 4.1). A Beneš network can implement an arbitrary permutation of its input channels [24] without blocking. This approach means that I can generate any permutation of the input/outputs and perform simultaneous data transfers using that permutation through the network with constant latency.

A Beneš network takes $2 * \log N - 1$ stages and $2 * N * \log N - N$ switches [24] where

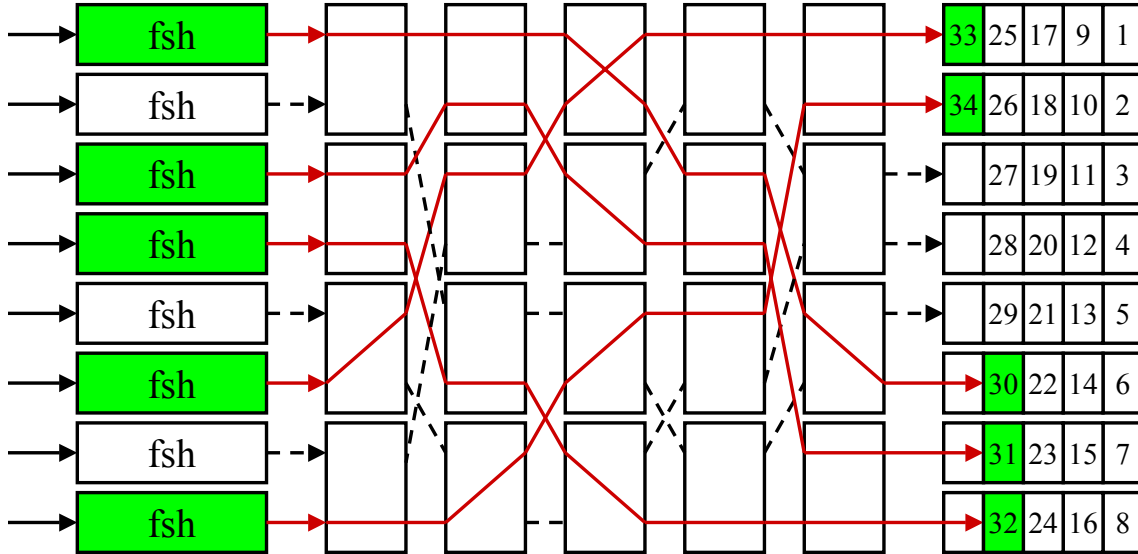


Figure 4.1: Hardware to support stream compression. This one is used to connect the output of 8 parallel fragment shader units to an 8-way striped memory system. The network is used both to compress out gaps in the five records being written (in this example) and to align the records being written with the striping pattern of the memory.

N is the number of programming elements (PEs). For a typical configuration with 8 or 16 PEs, the implementation of a Beneš network on the GPU requires 40 and respectively 120 switches, a modest hardware cost compared to the rest of the GPU.

Using a Beneš network for memory I/O, null records can be skipped and the valid records can be “packed on write”, achieving packing at a small constant cost (Figure 4.1). This approach attaches the network to one bank of memory or register file. However, in a typical branching instruction, the records are not killed, but rather routed to different buffers. A refinement of this basic approach, therefore, is to attach the Beneš network to two memory banks or register files. The records are classified into two categories based on the control flow and are simultaneously packed and routed accordingly. This achieves true

branching at virtually no execution time cost. Alternatively, I can write to the different targets in different clocks and just need the ability to rapidly redirect the output of the entire network to a different portion of memory.

4.1.2 Stream Packing on Current GPUs

Since hardware features vary from model to model, I'll restrict the discussion to two groups of GPUs that were considered to develop the system: The NVidia GeForce FX series (Ge Force FX 5200, 5600 and 5900) and ATI Radeon series (Radeon 9700, 9800). I refer to these cards generically as NVidia and ATI GPUs respectively.

The packing operation can be explicit, like in the case of conditional streams or Beneš networks, where data is moved around to compact the stream, or implicit, where the data does not move (or the move is opaque to the user), but there is a mechanism to mask out and avoid computations on null records.

All graphic cards support some form of conditional masking of fragments. The earlier GPUs employ a stencil buffer that masks out regions of the screen. Controlling the stencil buffer bit, one can control the data written into the buffer. More recent graphic cards also support conditional assignment and conditional fragment kill operations. Inside the fragment shader, an output variable can be given a specific value based on a data-dependent condition or, in the case of fragment kill, an element can be discarded based on a data-dependent condition. The fragment kill support on current GPU has two important limitations:

- It does not necessarily improve performance: the SIMD execution style employed results in idling the fragment unit until the rest of the adjacent fragments finish their

execution.

- It does not remove redundant space from the stream. For example, killing all fragments but the first and the last, results in a sparse stream that is stretched across the same space as the original one.

Some GPUs have support for “early stencil test” or “early depth test”. The “early” tests allows the GPU to avoid redundant computations on null records by masking them early in the pipeline and using a load balancer to re-distribute the work load. However, for this to work, the stencil or depth buffers need to be set up in a previous pass.

The early decision implies additional passes and the tests are not 100% percent accurate: not all null records are eliminated, because a lower-resolution stencil or z-buffer is used for the early test. These lower resolution buffers effectively break the test into spatially coherent tiles, and work is only avoided if all records in a tile are null. The success of the early tests rely on the distribution of the null records. While for a typical graphical application, the performance gain using early tests is considerable, for general purpose stream data this might not be the case.

4.1.3 Unpacking

The packing algorithm is conceptually simple: sparse data in a buffer is moved sequentially to the beginning of the buffer resulting in a smaller contiguous block. The problem is that the original position of the stream elements is lost. In most cases, the stream elements have to be moved back to their original position later. This step can be done efficiently on current hardware as described in the following sections (Fig. 4.2).

The information required to reconstruct the original stream is built and stored on the card. I call this generically a *reconstruction map*. I developed two ways to implement the reconstruction map:

Data Scattering: the original position of a fragment must be stored in the data stream.

When required, the original stream can be reconstructed by scattering the data using the original location stored with the stream elements. This approach minimizes memory usage, and it is conceptually simple. Data scattering support on GPUs is described in section 2.3, but basically requires feedback through the vertex unit that is currently poorly supported.

Permutation Map: This representation maintains a separate full size map with the original positions of all fragments. The stream is reconstructed by rendering a large quad and using the permutation map as texture coordinates in the packed stream. This approach is not as efficient as the data scattering approach since a larger number of fragment than necessary are processed, but due to limitations with vertex feedback on current GPUs, this is the method that we implemented. Therefore, it is discussed in detail in the following sections.

The packing operation behaves well under composition. An already packed stream can be packed again by composing the old reconstruction map with the new reconstruction map. However, this discards any intermediate information. An alternate solution is to maintain a stack of such maps. However, maintaining a map stack is too complicated and memory demanding and my results showed that streaming graphs can be optimized to avoid unpacking until the very last stage of the algorithm. Therefore, the system maintains

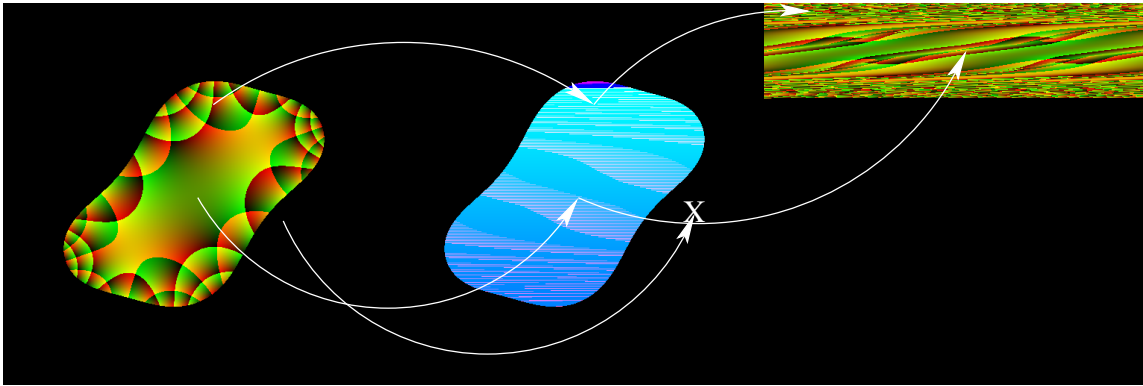


Figure 4.2: Left: Sparse stream, middle: permutation map, right: packed stream.

only one map, which is used to reconstruct the order of the original stream.

Packing, unpacking and other buffer management operations require precise information about the streams: marking discarded pixels, permutation maps, etc. I store all this information separately into a control channel and a permutation map, respectively. I keep one control channel and one permutation map per stream.

The permutation map (Fig. 4.2) is a fixed size texture that uses the first two components to map each texel to its offset in the stream and the third coordinate to mask out invalid fragments. To reconstruct the stream, the reconstruction map encodes the offset in the packed stream and it copies the record from there. If the reconstruction map does not point to anything (the black areas in the picture) it means that the corresponding record is null.

The control channel stores state information about the stream data. The control channel, unlike the permutation map, has to be the same size as the stream. Currently, the control channel uses only one component to mask out invalid fragments. I need a fragment mask for both the permutation map and the data stream because the algorithm does

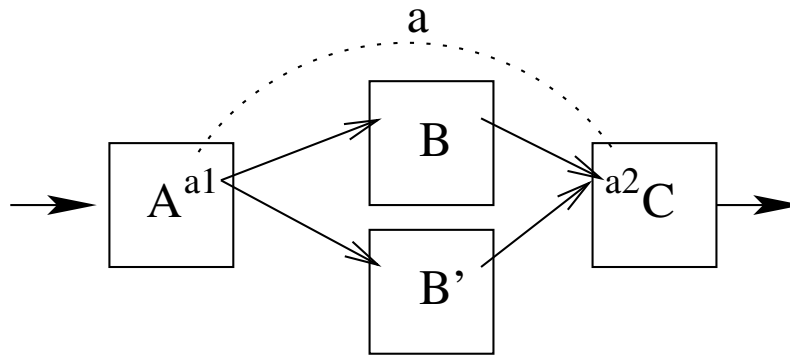


Figure 4.3: Resolving dependencies by adding new variables.

not need packing at every iteration. Therefore, when packing/unpacking is performed, masking bits from both the permutation map and from the control stream are used to mask out fragments.

To resolve stream size queries, the system uses the OpenGL extension *GL_ARB_occlusion_query*. This allows the schedule to determine how much non-null data exists in an output stream after a conditional write, and allows the scheduler to make a dynamic decision about scheduling and packing. This extension is easy to use and commonly supported on current GPUs. However, the overhead associated with it includes an additional rendering pass and some data transfer between the host and GPU memory. This step can be an expensive operation. Some of this cost can be amortized if this extension is used asynchronously.

4.2 Graph Structure

The transformation module takes the control graph output from the Sh compiler in a standardized format and transforms it into a format that our scheduler can operate on and that is suitable for a multi-pass approach. The main difference between the two formats is that the entire control graph output by the compiler represents semantically one program while in a streaming graph *each node* represents a separate program that can be independently run with no knowledge of the global context. Several issues need to be taken care of to convert one form to another. In particular, the system needs to resolve dependencies between adjacent nodes. Temporary data needs to be carried “forward” or “backward” from pass to pass. For example, suppose a variable a in the original program is used in both kernels A and C . Variable a is mapped to two different variables $a1$ in kernel A and $a2$ in kernel C as illustrated in figure 4.3. There is no arc in the graph from A to C , but there is a path that goes through B or a path that goes through B' . The data from $a1$ has to be passed to $a2$, so B has to be augmented. This procedure should be done only if B has a conditional output; however in my experimental prototype I do this regardless if B has a conditional output or not. Some of these scheduling problems are difficult to solve efficiently. Therefore, I imposed a few restrictions that, while they may affect performance, do not diminish the scope or correctness of the algorithm.

The first restriction is that I allow at most two output arcs (one conditional and one unconditional) and at most two input arcs. While this effectively forbids direct translation of certain high-level constructs like *case* statements, at a semantic level any graph can be reduced to an equivalent one that has at most two arcs coming in and two going out. The second restrictive decision, perhaps with more impact on performance, is my implementa-

tion of synchronization. As illustrated above, some temporary variables used in more than one node of the graph need to be synchronized and passed from one node to the following nodes maintaining temporal coherence and data consistency. One solution is to label all variables that are used in more than one block and to carry them in separate channels. Such a variable is called a *shared variable*.

The final version of the streaming graph is structured as follows: the entire graph contains exactly one entry node and one exiting node. The arcs represent streams of data passed from node to node. Under the above mentioned assumptions, all arcs have the same structure: an ordered sequence of data channels for each shared variable. If a variable is not used in a given node, it is automatically passed through. Buffer allocation is done automatically by the system at the end of the pre-processing stage. Each node has a program associated with it that is ready to be bound and executed. Since these programs have input and output signatures dictated by the shared variables, the entry and exit nodes have two additional programs associated with them that correspond to the global program's input and, respectively, output signatures. These programs are called *secondary programs*.

4.2.1 Node Types

On the simulator, kernel output is written into a stream register. The stream register does not have to be empty when the data is written; data in the stream register can be accumulated over multiple passes as illustrated in figure 4.4.

In the case of GPU frame-buffers, there are two cases: with packing and without packing. The case where there is no packing is similar to the simulator. Data can be accumulated over multiple passes (fig 4.5). The only difference from the simulator is that the data

streams are not compact anymore. Therefore, the user has to guarantee that it does not overwrite previously written data because in this case the new stream is not appended, but rather interleaved with the old one.

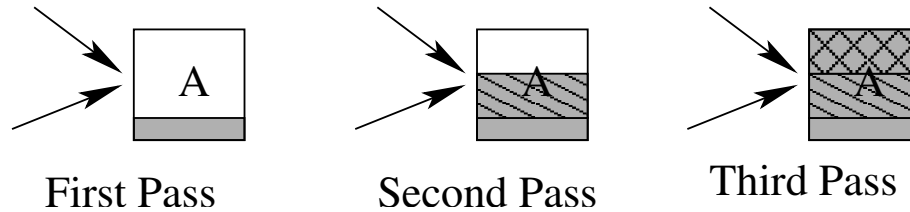


Figure 4.4: In the Sm simulator, data in the stream registers can be filled sequentially over multiple passes.

However, when packing is done, it is trickier to maintain data coherency. Recall that processing a packed stream is done by rendering two adjacent quads. This method works under the assumption that the rendering buffer is empty. If the buffer is not empty, these two quads overwrite previously written data. One solution to this problem is to render a different set of quads that tiles a contiguous block of data adjacent to the one already there as illustrated in fig 4.6. This technique works only if the target buffer is packed and querying the size is required. It also makes the buffer management and packing more complicated. Therefore, for the purpose of this prototype, a kernel can be scheduled only if its output buffers are empty.

This assumption may lead to redundant passes due to fragmentation of the original stream. To address this problem the system allows explicit merging operators. For example, figures 4.7 and 4.8 show a snapshot of a scheduling process. The programs have four kernels and at each pass data flows from the first kernel (A) to the last kernel (D). In figure 4.8, we explicitly merge the data at node D . That is, we wait for all the data from

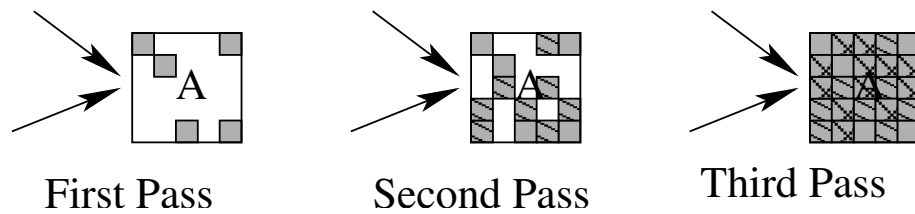


Figure 4.5: On the GPU, with no packing, the frame buffers or P-buffers can be filled over multiple passes. In this case the streams can be sparse.

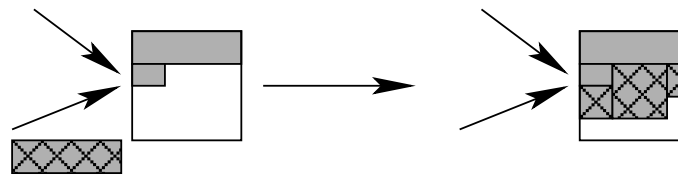


Figure 4.6: Appending data to a packed buffer by rendering three quads.

all the input branches to arrive before scheduling kernel D . This only takes 5 passes. In figure 4.7, scheduling with no explicit merging takes 7 passes, because kernels D and E are scheduled twice. I refer to this case as a *spatial merge*.

If data is packed, even if there are 5 or 7 passes, the total number of processed fragments is the same in both cases. The gain is only in the overhead associated with kernel switching. This overhead can be small, if it is simply a state change, or it can be large if additional data has to be copied into the GPU memory.

Similar optimizations can be found for iterations as well. Data exiting a loop can be merged together. I refer to this case as a *temporal merge*. Figures 4.9 and 4.10 illustrate this. In both cases there are three kernels: A represents the condition at the head of a loop, B represents the body of the loop and C represents the code that is executed after the loop is completed. In figure 4.9, scheduling with no explicit temporal merging takes 8 passes

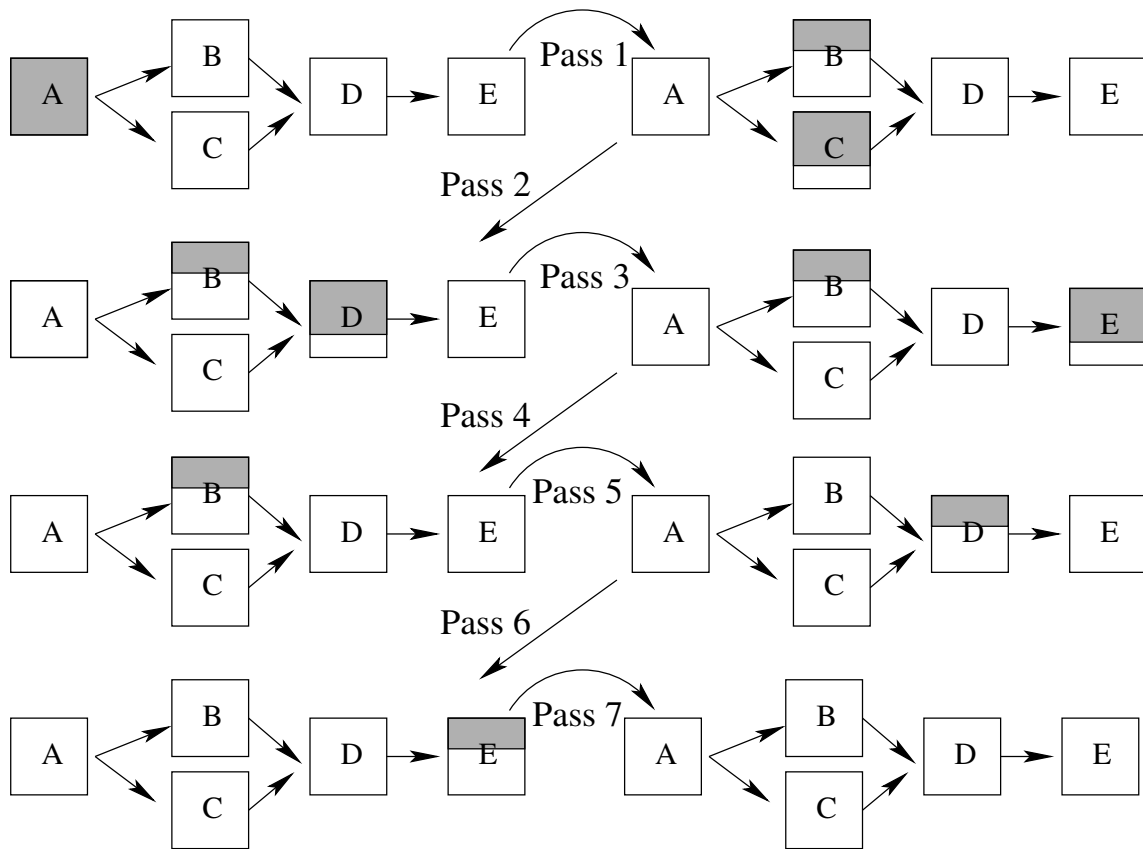


Figure 4.7: A correct scheduling sequence with no synchronization takes 7 passes to solve.

while in figure 4.10 with explicit merging it takes only 6 passes.

These merging operation are done using some special types of nodes in the graph as described in the following paragraphs. An analysis of these nodes is provided in section 4.4.2.

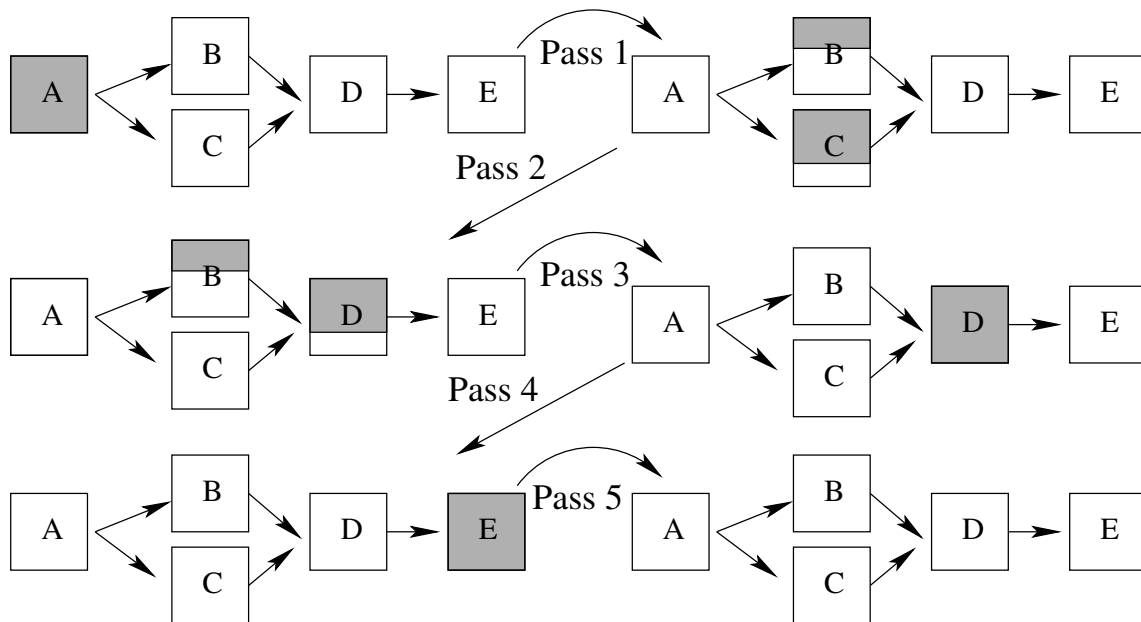


Figure 4.8: A correct scheduling sequence with synchronization on merge takes 5 passes to solve.

A streaming graph has three node types:

Accumulation Node. It is used for explicit temporal merging. In a “while” loop structure, some data may complete the loop on any iteration. This node is used to block the data flow until it accumulates all data that entered the loop. The accumulation node has special properties: it has exactly one predecessor and only one follower and it runs in two beats: the first beat is right after its parent so it can cache the data that escaped the loop and the second beat is when all data in the loop is consumed, it synchronizes and releases it for further processing.

Merging Node. This node synchronizes two streams that have been previously split by a branch. Unlike the accumulation node, this is a spatial synchronization point where

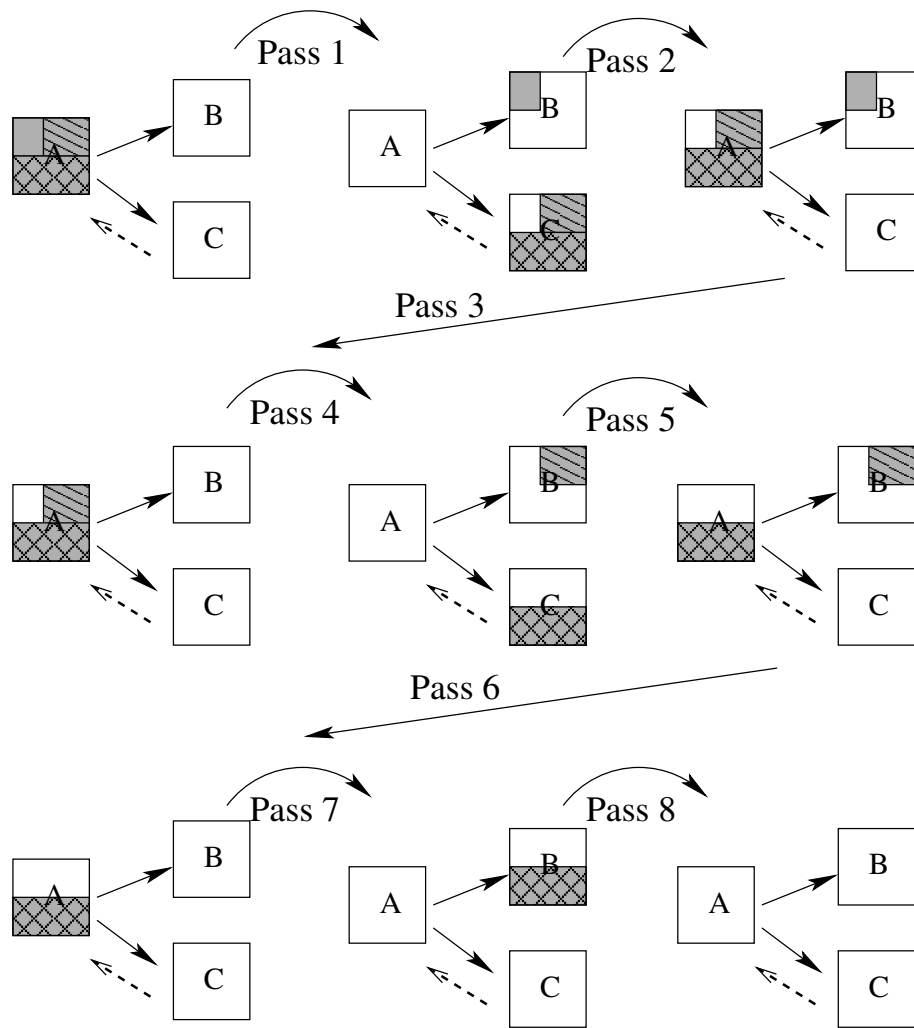


Figure 4.9: A correct scheduling sequence with no synchronization takes 8 passes to solve. Packing is omitted for clarity.

data from two streams are interleaved based on the original ordering. The merging is based on the control channel data. The branching operation has the property that at most a fragment in any given position is valid in exactly one control stream. However, this implies that unpacking is required before or while a merge node is

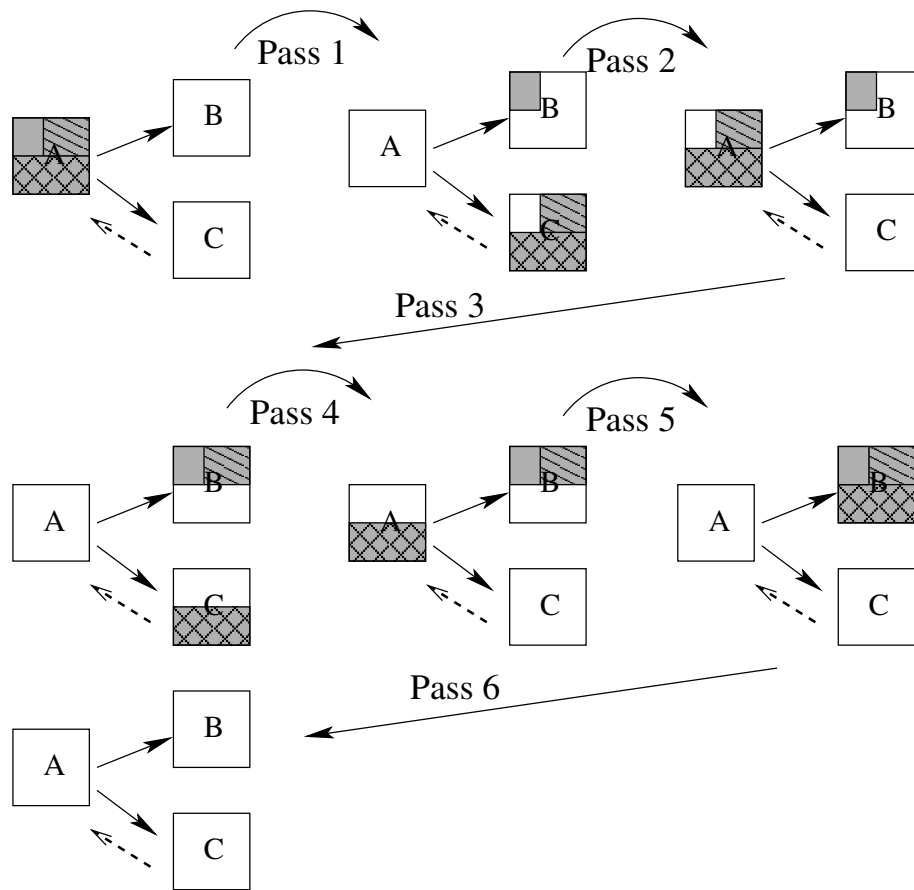


Figure 4.10: A correct scheduling sequence with synchronization takes 6 passes to solve. Packing is omitted for clarity.

executed. This node has two inputs and is not restricted on the number of outputs. This node is also optional used by the scheduler for optimization purposes similar to the accumulation node mentioned above.

Regular Nodes. A node that is neither an accumulation node nor a merging node.

The accumulation nodes and merging nodes are optional. A graph can be constructed from regular nodes only. The purpose of the accumulation and merging nodes is to perform

explicit merging for optimization purposes.

Any node in the graph (with the exception of the accumulation node) can be marked as a branching node. These nodes split the output data based on a conditional variable and send it over two different arcs. While my results work on an arbitrary control graph subject to the restrictions mentioned above, the compiler generates only two non-trivial¹ subgraph types: branching and looping (Figure 4.11).

4.2.2 Graph Transformation Algorithm

The graph transformation algorithm has the following stages:

Build a Variable Map: Traverse the program graph and record for each variable what blocks it is used in and for what purpose: read, write or conditional.

Create the Individual Programs: Take each basic block and transform it into an independent program:

1. Traverse the statement list, and for each variable create a new one that has the correct scope (e.g. input, output, const, temp, etc.) and the correct state (e.g. swizzle, data, etc.).
2. Copy all statements using the newly created variables.
3. Add the unused shared variables to the list of inputs and outputs so the system passes them forward.

¹A trivial graph is a graph that has a linear structure.

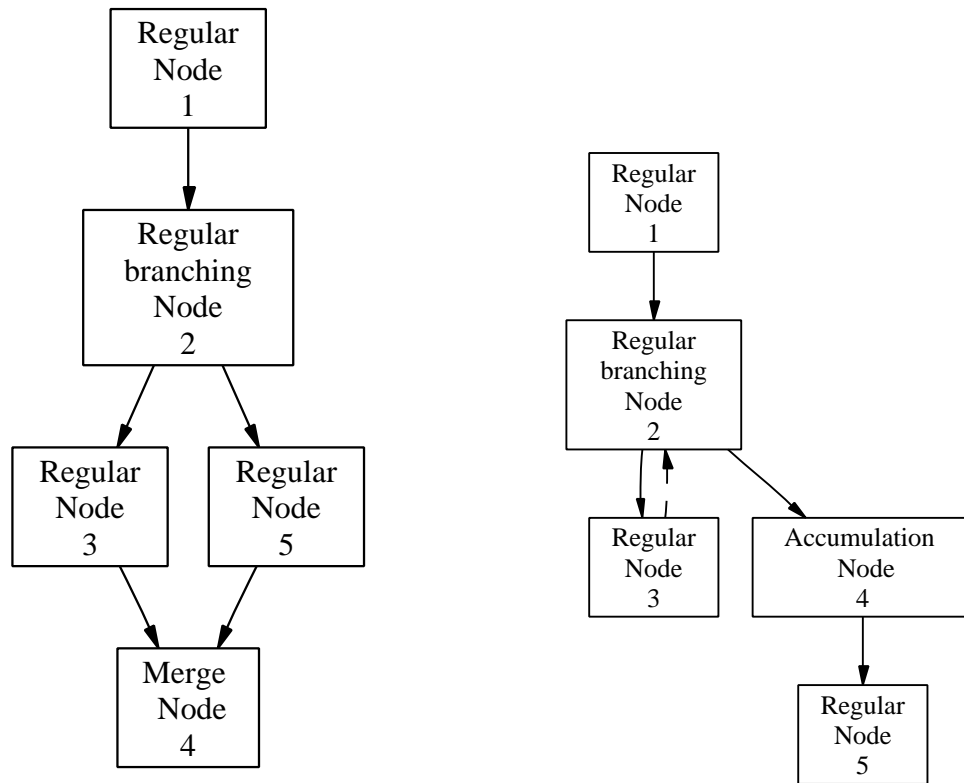


Figure 4.11: Left: graph generated by an **SH_IF/SH_ELSE/SH_ENDIF** statement; Right: graph generated by a **SH_WHILE/SH_ENDWHILE** statement. All regular nodes that do not have a follower (or a predecessor) can be connected to another graph downstream (or respectively upstream)

4. Order the list of input and output variables so they match the input and output variables of the adjacent nodes. The consistency of the input and output variables is achieved by imposing a global order on shared variables.
5. Add additional inputs and outputs for the control stream.

6. Append or prepend snippets of code as needed. These pieces of code used to glue the programs together are called *auxiliary programs*. This transformation is done using shader algebra operators as described in section 2.5. These are the situations where this is required:
 - Branching. Appends a program that conditionally writes the data into a different stream.
 - Loop Merging. Loop merging occurs when the data escaping a loop is accumulated. A special node is inserted that acts like a barrier. This node is the only node that is not based on a former basic block, and therefore, it is not created using the shader algebra.
 - Regular Merging. Prepends a program that chooses the valid fragments from two streams based on their control channels. I established before that I can guarantee that they cannot both be valid fragments.

Figure 4.12 describes steps 5 and 6 in more detail.

7. Repeat steps 2-5 for the secondary programs if applicable (e.g. this node is an entry or exiting node).

The input and output variables need to have a global ordering, so I created the following ordering convention:

1. Conditional variable, if applicable. No conditional output variable exists in the final form of the stream graph, but since the graph transformation are made in independent stages using the algebra operators, some intermediary programs might have conditional input/output variables.

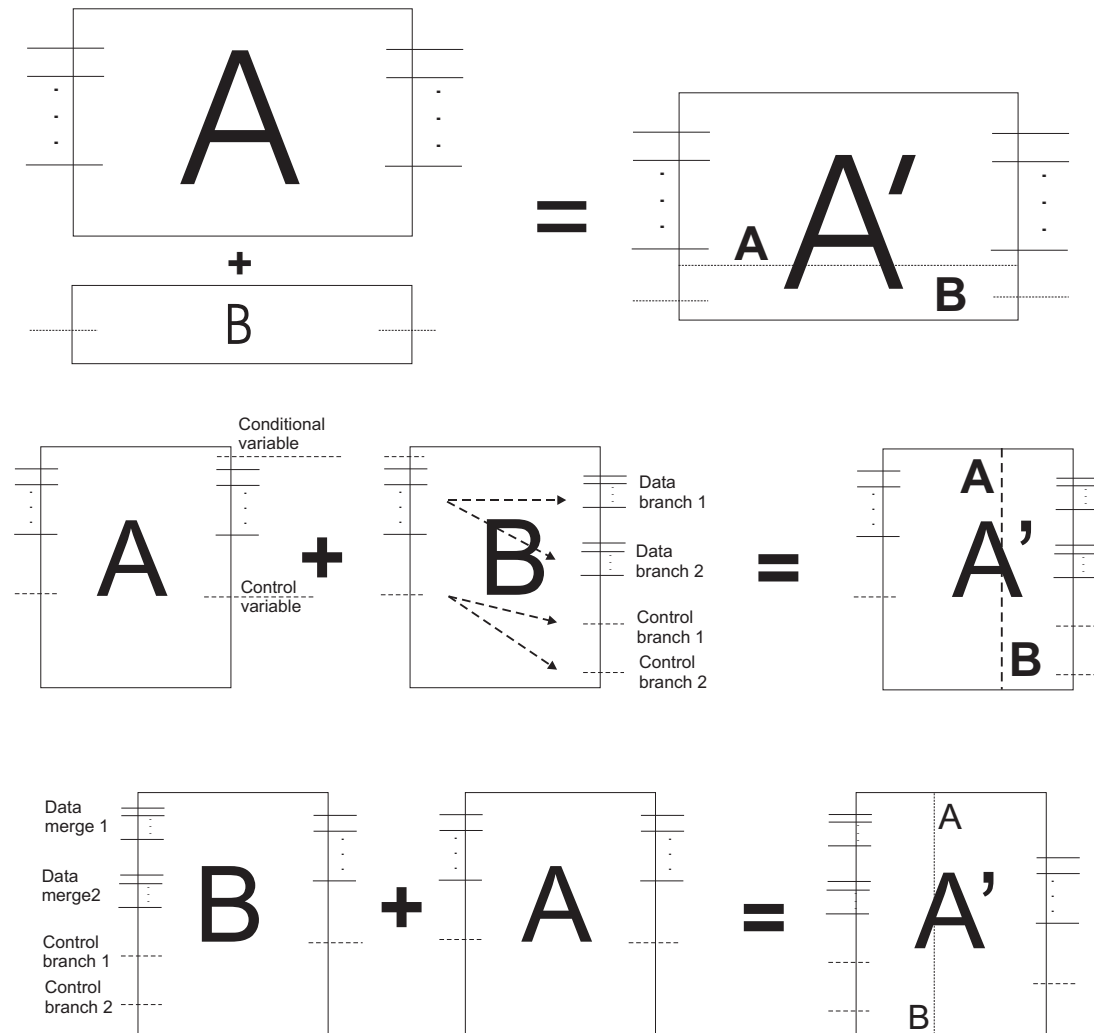


Figure 4.12: Describes how the shader algebra is used to add data-dependent control flow support to kernels. Top: attach a pass-through variable. Middle: append a program that branches the input based on a conditional variable. Bottom: prepend a program that merges two streams.

2. Shared variables. Data passed from one node to the next. The shared variables are subject to global ordering.

```

ShProgram ifp = SH_BEGIN_PROGRAM("gpu:stream"){
  ShInputTexCoord3f input; // the coordinate of the point scaled between [0, 1]
  ShOutputColor3f ocolor; // output color

  ShAttrib1f d = 0.1;
  ShAttrib3f s;

  s(0)= (floor(input(0) / d(0)) + floor(input(1) / d(0))) * 0.5;
  s(1) = floor(s(0));

  SH_IF(s(0) - s(1) > 0.25){
    ocolor = ShAttrib3f(0, 0, 0);
  } SH_ELSE {
    ocolor = ShAttrib3f(1, 1, 1);
  } SH_ENDIF;
} SH_END_PROGRAM;

```

Listing 4.1: Checkerboard code

- Control variables. The current implementation has only one control variable that is used as a mask to determine if the record is valid or null. However, if this is a branching node, there is one control variable for each output stream.

If we have multiple incoming or outgoing streams, the stream ordering has higher priority. For example, if the current configuration has three shared variables, *normal*, *tangent* and *color*; the output of a regular node is: *normal*, *tangent*, *color*, *control-variable*. If this node is a branching node, the output variables are: *normal1*, *tangent1*, *color1*, *normal2*, *tangent2*, *color2*, *control-variable1*, *control-variable2*.

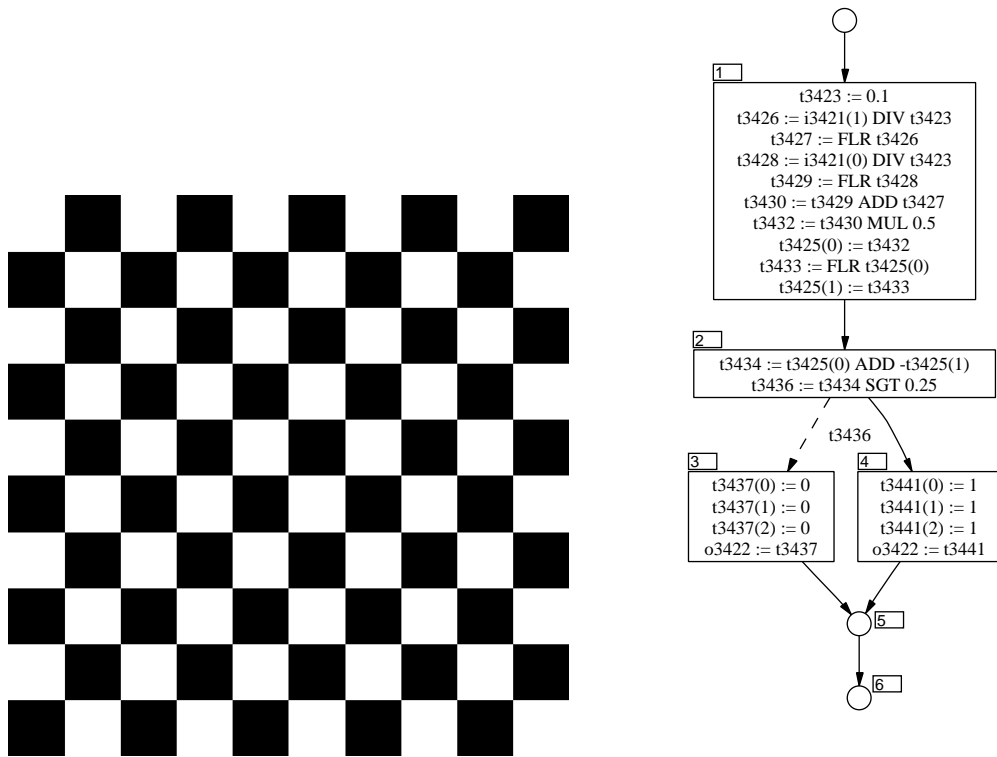


Figure 4.13: Checkerboard and its program graph output by the Sh compiler

4.3 Examples

I illustrate my algorithm using two simple examples. The streaming graphs for these examples are similar to the ones in figure 4.11

4.3.1 Checkerboard

Consider the program from Listing 4.1 that renders a checkerboard. The graph output by the compiler can be seen in Figure 4.13.

The algorithm proceeds as follows:

1. It scans all the nodes of the graph for shared variables. It marks the variable $t3425$ and $o3422$ as shared variables. The variable $t3425$ corresponds to s and $o3422$ corresponds to $ocolor$ variables in the original program. The variable $t3425$ is used in blocks 1 and 2 and $o3422$ is used in blocks 3 and 4. The program also marks the variable $t3436$ as a conditional variable.
2. It generates the kernels: new independent programs for each of the blocks. The shared variables marked in the above step become input/output variables in the new programs, which ensure the data gets forwarded. The conditional variable becomes an output variable that is appended to a branching auxiliary program. If the conditional variable is also a shared variable, the transformation has to duplicate the variable, because a copy is needed to be passed forward.
3. Since not all shared variables are used in all blocks, but they need to be passed forward, I iterate through the kernels and add each unused shared variable to the lists of inputs and outputs. This situation is demonstrated better in the Julia set example. In this simple example, this operation is unnecessary as data travels at most one step. Even in this case, the algorithm still copies the data because in this prototype no complete dependency analysis is made. In theory, a further optimization step could eliminate some pass-through operations in the absence of conditionals.
4. When executed, the data flows from one kernel to the next. Therefore, the order of the outputs of a kernel has to match the order of the inputs of the following kernel. A global ordering is forced: in this case $t3436, t3425, o3422$.

5. The raw kernels generated based on the basic blocks of the original graph are further processed to accommodate the data flow. An auxiliary program is appended to kernel 2 to split the data and kernel 5, since it does not have any code, is replaced by an auxiliary program to merge the data. It should be noted that the computation performed by this kernel inside the conditionals is far too simple for this transformation to be worthwhile, and in practice conditional assignment would be more efficient. However, the example is presented like this for illustration purposes. It should be kept in mind that this transformation is most useful when the computation guarded by the IF statement is expensive or potentially non-terminating.

4.3.2 Computing the Julia set

The Julia set example tests the looping support of the system. Listing 4.2 shows the Sh code used to produce the Julia. Figure 4.14 shows the control graph of the Julia set and figure 4.15 shows the results.

The algorithm proceeds as follows:

1. Scan all the nodes of the graph for shared variables. Variables $t3423$, $t3424$, $t3428$ correspond in the high level code to the variables pos , i and c respectively are the shared variables. Variable $t3451$ is marked as conditional variable.
2. Generate the kernels.
3. Iterate through the kernels and add the unused shared variables to the list of inputs and outputs. Variable c (alias $t3428$) is used in blocks 1 and 3. Block 2 does not use

```

ShAttrib1f iterations = 5.0; // number of iterations
ShAttrib1f color_scale_factor = 1.0 / (iterations+1.0);

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
    ShAttrib3f c(-0.122, 0.745, 0.0); // julia set constant

    // one 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 = i(0, 0, 0) * color_scale_factor(0, 0, 0);

SH_END_PROGRAM;

```

Listing 4.2: Julia set code

it, so it does not appear in the kernel associated to block 2. Therefore, the system must add it artificially to maintain data coherency.

4. The global ordering in this case is: $t_{3451} < t_{3423} < t_{3424} < t_{3428}$

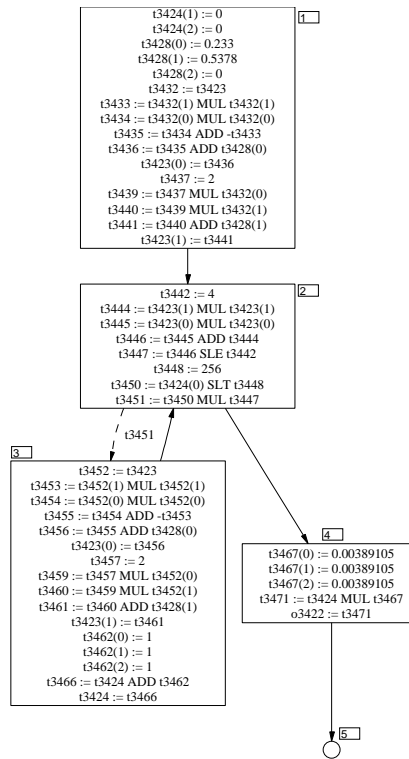


Figure 4.14: Compiler control graph for the Julia set

- In this stage, auxiliary programs are glued similar to the previous example. The only exception is the accumulation node. The accumulation node is inserted in the graph between nodes 2 and 5 (Figure 4.14)

4.3.3 Combined Example

To show a more complex example, I combined the checkerboard example with the Julia set example (figure 4.16). The streaming graph in figure 4.17 illustrates the high level structure of the program. The program consist of two iteration loops nested inside a conditional.

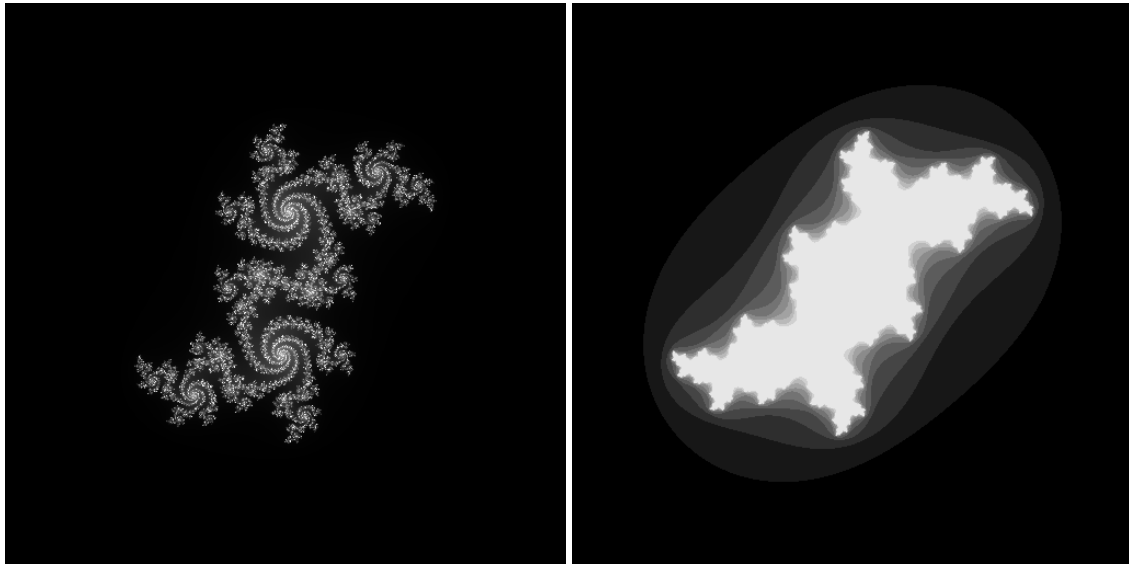


Figure 4.15: Julia set examples. Left: 256 iterations with $c = (0.233, 0.5378, 0)$. Right: 10 iterations with $c = (-0.122, 0.745, 0.0)$

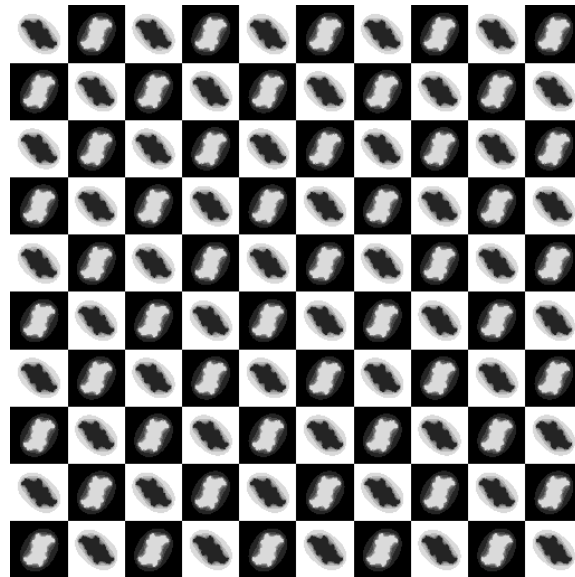


Figure 4.16: This shows a combination of the Julia set and the checkerboard example.

The conditional construct corresponds to the checkerboard and the iterations correspond to the Julia set.

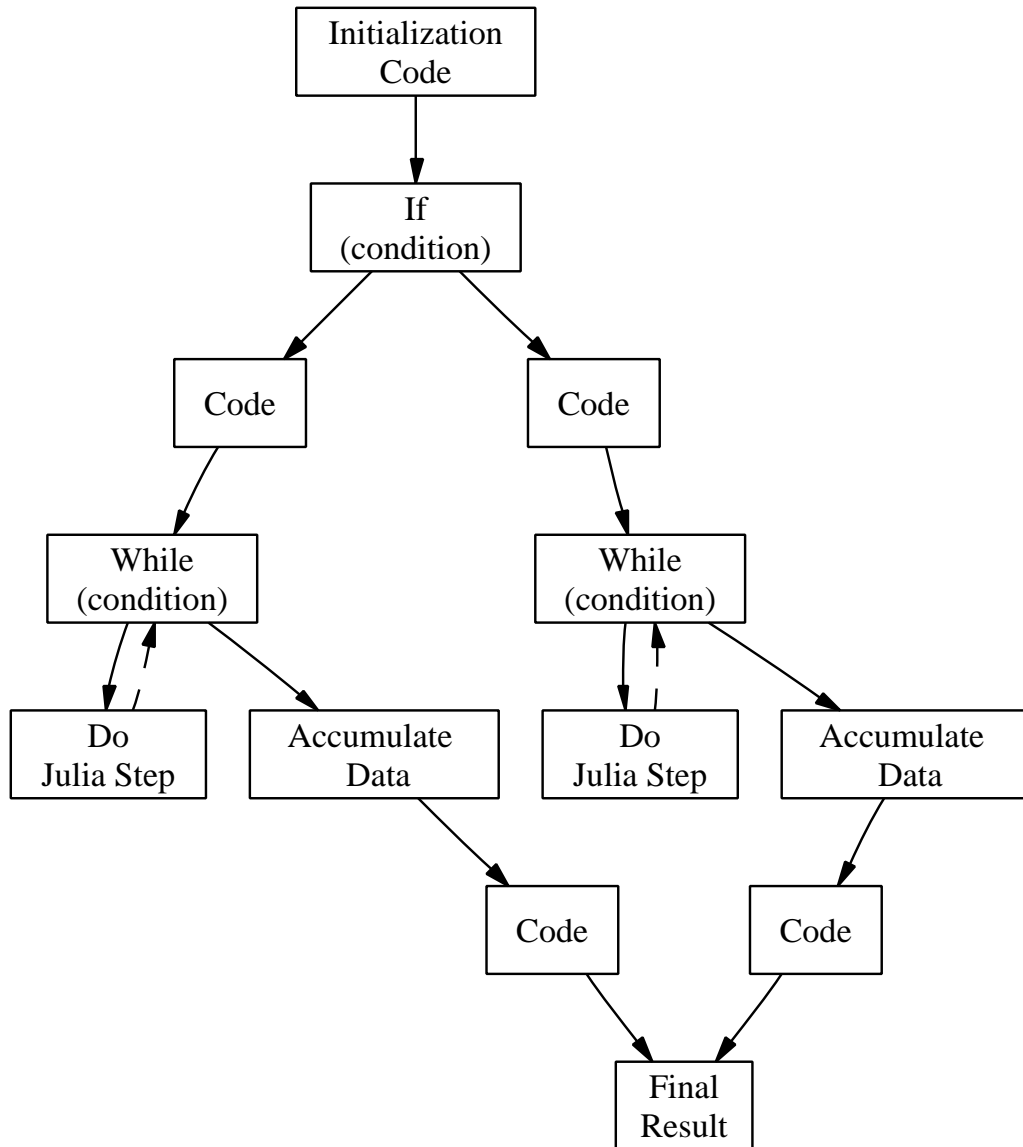


Figure 4.17: Schematic streaming graph of the checkerboard Julia set example.

4.4 Scheduler

4.4.1 Scheduling Algorithm

The scheduler takes a streaming graph, an input stream and an output stream. It executes the graph until all inputs are consumed and all results are written to the output stream. First, I allocate intermediate buffers that are used to facilitate the data flow from one kernel to the next. Therefore, each arc has some GPU memory associated to it. Then, for each pass, the scheduler chooses the next running kernel from a pool of active kernels. A kernel is active if it has input data to consume and its output streams are empty. If a kernel is not active, it is either in a state of starvation (no input data) or congestion (it has no space to write the output). Table 4.4.1 formalizes these conditions for the various node types. Consider first the case where the scheduler does no packing or unpacking

	Conditions for Data Starvation	Conditions of Data Congestion
Regular Node	all input streams are empty	any of the output streams are full
Merging Node	any of the input streams are empty	any of the output streams are full
Acc. Node	any of the input streams are empty	N/A

Table 4.1: Node types and their availability conditions

operations. The control graph together with the current history of kernel executions give information about which kernels can execute at a given point in time. In fact, if the control graph has no loops, a static schedule of kernels can be created. If the control graph has loops, a static schedule is not sufficient and the system has to rely on querying the size of the streams to find a correct scheduling sequence, where a correct scheduling sequence is a finite sequence of passes that yield the expected result in the output stream. For a

general program that uses data-dependent iterations, a combination of static flow analysis and dynamic scheduling is required. I have considered so far the easier case where there is no packing. Packing increases significantly the complexity of the problem. Even simple programs with branching and no loops can have different performance depending on the scheduling order and the structure of the streaming graph.

The scheduling algorithm goes as follows:

1. All kernels maintain an execution count that is initialized to zero. As a preprocessing step, a breadth first search tree is run and all arcs are marked as being “forward” or “backward” based on whether or not they belong to the tree. The backward arcs correspond to the “feed-back” arcs in a loop structure. When a kernel runs, its count is incremented with the exception of the accumulation nodes. The accumulation node is blocked until the opposite arc of its parent has size zero. This indicates that the loop is exhausted. The accumulation point is scheduled next and its count is set to the count of its parent.
2. From the pool of active kernels, choose one based on a heuristic (to be described later).
3. Unpack the incoming streams if necessary.
4. Run the kernel.
5. Pack the output data streams if necessary.
6. Update the pool of active kernels.
7. Repeat steps 2-6 until the pool of active kernels is empty.

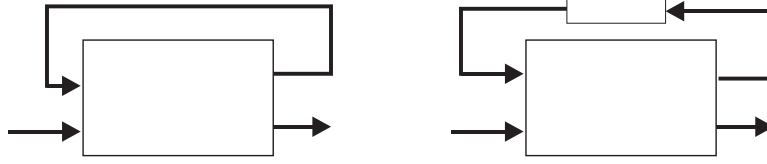


Figure 4.18: The system avoids dead-locks by creating artificial nodes

A kernel is active if it is neither in a state of starvation nor in a state of congestion. If q , p and f are nodes and ℓ is an arc, let $size(\ell)$ be the size of the stream associated with that arc, $c(p)$ denote the execution count of p and $back(p, q)$ be a function that returns true if the arc from p to q exists and it is a backward arc and false otherwise. If q is a accumulation node, by construction it has only one predecessor and the predecessor is a branching node having exactly 2 output arcs: one towards q and another denoted by $\mathcal{U}(q)$. If, in particular

- A regular node q is not starving iff $\exists p$ a predecessor of q , $(back(p, q) \wedge c(p) = c(q)) \vee c(p) > c(q)$
- A merging node q is not starving iff $\forall p$ predecessor of q , $back(p, q) \wedge c(p) = c(q) \vee c(p) > c(q)$
- An accumulation node q is not starving iff $size(\mathcal{U}(q)) = 0$
- A node is not congested iff $\forall f$ follower of q , $back(q, f) \wedge c(q) < c(f) \vee c(q) \leq c(p)$
- If a node is neither starving or congested, it is active and it can be scheduled for execution.

The packing/unpacking rules are as follows:

- Streams should only be packed following a branch. Otherwise, the packing has no benefit.
- Both branches going into a merge node need to be unpacked. A similar rule applies for arcs going into an accumulation node, which is a condition required to insure data consistency.²

To avoid dead-lock I do not allow self-looping (Figure 4.18). In the construction phase, a dummy program and a loop merge program with their associated buffers are inserted in the graph. This avoids having programs write to their inputs, an operation not permitted on GPUs.

4.4.2 Scheduling Strategies

There are two cases. In the first case, data is always packed on write and the overhead associated with it is negligible. In the second case, packing is a distinct operation with potentially significant overhead. This implementation assumes the first case and since current GPUs do not pack on write, and packing is simulated on the CPU. Unpacking, on the other hand, can be implemented on current GPUs using fragment programs (section 4.1.3).

To avoid redundant passes, the GPU implementation uses merging nodes and accumulation nodes whenever possible as described in section 4.2.1 . These types of nodes require data to be unpacked before processing. Figure 4.19 illustrates why this is the case.

²Note that these nodes are optional and they serve only optimization purposes.

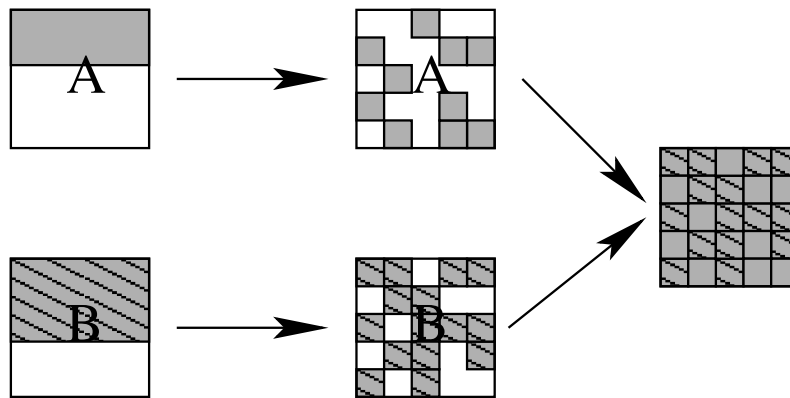


Figure 4.19: Spatial merging of packed stream is done in two stages: unpacking and then rendering one on top of the other.

Merging two streams on the GPU is done rendering them in sequence. In order not to overwrite records, a valid fragment in the first stream must be invalid in the second stream and vice-versa. This can be guaranteed only if the streams are unpacked.

In some cases this leads to additional unpacking steps that may degrade performance. However, on a processor that supports “packing on write”, this problem does not apply. On such a platform, the data is sequentially written to a buffer that is not necessarily empty, resulting in free merging³.

³Without preserving the order.

4.5 Performance Analysis

4.5.1 Cost Models

The cost model of an application running on a GPU has to include the overhead of transferring data to and from the GPU memory. For current GPUs, this transfer has relatively high cost and it is more efficient to transfer data in large blocks rather than a few records at a time.

Buck et al. [3] present an analysis of the computational advantage of CPUs vs. GPUs on a streaming computational model. Their results show that for small streams, GPU running time is bounded by the transfer time. It is only for large enough streams that the actual computations dominate the running time. The relationship of one computational advantage over the other is thus defined as the ratio between the number of computations and the size of the transferred data. This ratio is called *arithmetic intensity* and is denoted by α . According to their results, GPU outperforms the CPU if:

$$\alpha > \frac{R^{-1} + W^{-1}}{K_{cpu}^{-1} - K_{gpu}^{-1}} \quad (4.1)$$

where R and W are the bandwidth rates of reading from and respectively writing to the GPU; and K_{gpu} and K_{cpu} are the execution rates for the GPU and for the CPU, respectively. As defined above, on a given architecture α should be a constant. However, the transfer rates depend on the size of the data transferred. Roughly, on GPUs similar to what I used, for streams between 10,000 32-bit floating point data (floats) to 100,000 floats, the α cutoff is 100 and for streams of over 1,000,000 floats, the α cutoff is about 40.

Therefore, to give the GPU a computational advantage on a stream containing 10,000 floats, a program of at least 1,000,000 instructions is needed, a large size for a program running on current GPUs.

Most of my testing programs have under 10,000 instructions with a data set of 500,000 floats; therefore, it is not surprising that the running times for my programs are larger than if they were executed on the CPU. Our goal however, is not to compare GPU and CPU speeds, but rather to verify the hypothesis that stream packing improves overall performance. Another factor not included in their analysis is kernel switching. This aspect is important because on current GPUs, the maximum size of a fragment program is relatively small (1024 instructions on NVIDIA GeForce FX cards and 64 instructions on ATI RADEON 9700-9800); therefore, a large program may require the number of passes to be in the order of thousands. In these cases, even a small overhead can be amplified into a significant delay.

Without creating a formal cost model, my performance analysis decouples the implementation layers, analyzing one implementation layer at a time. My algorithm operates on the stream abstraction available in Sh and it uses it opaquely. Should another stream implementation be available with similar functionality, this performance analysis will still apply.

4.5.2 Methodology

This implementation suffers from two important limitations. The first one is that packing is done on the CPU and its cost is large compared to the other stages of the algorithm. The second is due to the (*current, suboptimal*) stream implementation on Sh (*unnecessarily*)

copies the data between GPU memory and main memory for every pass. To make a plausible analysis, I need to make some assumptions related to these limitations. I believe that with some small extensions (section 4.1.1), packing can be done in the future on the GPU while writing fragments. This change would result in virtually free packing. Therefore, in this analysis, I factored out the packing overhead. To account for the second limitation, I ignore the GPU to host memory timings ⁴.

Since the implementation uses the Sh stream abstraction, we divide the performance analysis into two layers:

Low level layer: I timed the low level stream implementation.

High level layer: I timed various high level stream operations used in our algorithms (e.g. unpacking).

For profiling, I used a modified Julia set program. I appended to the loop kernel a large set of dummy instructions so the kernel has almost the maximum number of instructions allowed by the GPU (1024). I prevented the driver from optimizing the code by creating artificial variable dependencies. I ran the program on three buffer sizes: 64x64 (4096), 256x256 (65,536) and 512x512 (262,144) fragments. I varied the maximum number of iterations from 5 to 20. I executed each test three times and averaged the results.

4.5.3 Benchmarks

A stream execution has four major steps:

⁴Because I do not control the details of driver's memory management, I cannot factor them out completely.

GLX Context setup: creating an OpenGL context in which the stream execution takes place.

Fragment program setup: compiling and optimizing the fragment program. Since the GPU that I used for my implementation does not support multiple output buffers, Sh simulates this behavior by rendering multiple passes with a different output each time. The Sh optimizer performs dead-code removal at each step generating a different program for each of the passes. On a GPU that supports multiple buffers, this stage can be avoided by caching the pre-compiled programs.

Binding: downloading the program and its associated dependencies on the GPU (e.g. textures, matrices, etc.)

Rendering: actual data processing time.

The low level timing analysis of the system (Table 4.5.3) shows that the rendering and binding stages have almost equal running time. This similarity is because, at this stage of Sh development, streams are not optimized and redundant data is copied to and from the GPU. Both the fragment setup and binding stages can be optimized by a large factor once GPUs will have support for multiple output buffers and as the Sh streams implementation matures.

Resolution	GLX context	Fragment program setup	Rendering	Binding
256x256	3.62 %	23.40 %	38.19 %	33.19 %
512x512	2.27 %	11.48 %	44.79 %	40.85 %

Table 4.2: Work load distribution for Sh streams

At the high level, the execution of the algorithm has three major stages⁵:

Running time: Executing kernels.

Unpacking: Unpacking previously packed data.

Other: Miscellaneous overhead, independent of the packing strategy.

Table 4.5.3 illustrates the improvement in performance. The “Improvement” column shows the net improvement of run-time using packing. The “Improvement on run” column shows the theoretical speed-up of the execution time. The difference between the two is due to the unpacking overhead. The results show that the run-time performance of packing scales well with the size of the stream, showing increases in performance of up to 25%. The results meet our expectation since for large data sets the contribution of the constant overhead goes asymptotically to zero.

Table!4.5.3 outlines the various stages in the algorithm together with their distribution in terms of GPU/CPU time.

⁵packing is not included since we assumed it is free

Resolution	Iterations	Elapsed time when packing (ms)	Elapsed Time without packing (ms)	Improvement (%)	Improvement on run (%)
64x64	5	1796	1733	-3.5	1.9
64x64	10	3451	3360	-2.7	2.4
64x64	20	6723	6528	-2.9	2.5
256x256	5	2413	2588	6.7	16.1
256x256	10	4416	4950	10.8	19.7
256x256	20	8462	9655	12.3	21.6
512x512	5	4182	5391	22.4	35.1
512x512	10	7572	10223	25.93	39.58
512x512	20	14415	19925	27.65	40.94

Table 4.3: Performance improvements of packing

Resolution	Iterations	Run (%)	Unpacking (%)	Other (%)
64x64	5	88.64	4.85	6.51
64x64	10	89.33	4.98	5.68
64x64	20	89.56	5.23	5.33
256x256	5	83.17	7.92	8.91
256x256	10	83.47	8.42	8.11
256x256	20	83.42	8.82	7.76
512x512	5	76.20	12.20	11.60
512x512	10	74.83	13.03	11.54
512x512	20	75.10	13.75	11.15

Table 4.4: Work distribution when packing

Chapter 5

Conclusion

This thesis has presented a method to implement a stream processing computational model including data-dependent control flow on SIMD GPUs, more specifically on the fragment units. The method consists of a series of algorithms used to transform the control graph of a program output by a compiler into a stream graph, a structure suitable for multi-pass computations. A run-time scheduling algorithm is presented that takes the stream graph and determines the order in which the individual kernels run. I introduced the concept of packing, where the data of a sparse stream is rearranged into a contiguous block to avoid redundant computations. Two parallel control streams are generated to maintain data coherency and encode ordering information. I presented an analysis of packing and unpacking operations and how they integrate with the scheduler.

5.1 Systems Comparison

I described two implementations of this system: one running on Sm, a software simulator, and one running on top of Sh system and targeting real GPUs.

Sm was designed as an alternative architecture for next generation GPUs. Unlike current GPUs, Sm “packs on write” and it stores data streams ¹ in one dimensional buffers. Sm supports a limited form of recursion by allowing data amplification, and a hybrid depth and breath order for exploration of the recursion tree. A dynamic scheduler drives the simulator running kernels in sequence using a greedy based heuristic.

The GPU implementation uses the Sh system to drive the GPU. The most challenging problem on the GPUs was packing. Not only do current GPUs not support packing on write, but explicit packing on current GPUs proved to be difficult and inefficient. I provided a theoretical method to extend the current GPUs to support packing on write and analyzed the performance under this assumption while my prototype performs the packing on the CPU. I also discussed briefly various scenarios of packing on current GPUs.

5.2 Future Work

One of the bottlenecks in the system is redundant data transfer between kernels. A more comprehensive analysis of the streaming graph using algorithms and data structures borrowed from compiler theory should be used to sort out dependencies between variables and their associated data streams.

This system uses only the fragment unit of the GPU. The vertex unit can be used as a

¹vertex and fragment streams

vertex scatter engine for more efficient and less memory intensive unpacking. Even so the vertex unit would probably be idle most of the time. A better use of all GPU resources is desirable.

I currently pack data on the CPU which significantly reduces performance since the data transfer to and from the host is costly. An algorithm to pack data on the GPU has been developed, but due to limitations of the current graphics cards, the integration of GPU packing into our system was difficult. A hardware extension to current SIMD GPUs was presented that provides low cost “packing on write”.

While I presented a detailed analysis of various heuristics under the “packing on write” assumption, I did little to analyze the situation where packing is done explicitly at some potentially significant cost. The scheduling algorithm has in this case an extra degree of freedom deciding whether or not to pack. Since packing can be expensive, it can dominate the computation in a case where the stream has a small number of null records. The decision “to pack or not to pack” depends not only on the number of null records in the stream, but also on the computational cost of the kernels. For example, on the one hand, if there are 10,000 null records as input to a kernel that has only few instructions, the redundant execution might be faster than packing. On the other hand, if there are only 1,000 null records running a complicated program, the redundant execution might be slower than packing. Empirical heuristics can be developed in this case to make the packing decision.

My algorithm preserves the order of the data in the stream. However, there are cases when maintaining the order is not required. There are also cases where in the final stage of the algorithm the order is required, but not for intermediate steps. Such cases can yield optimizations. More efficient packing algorithms that do not maintain ordering might be

useful for such cases. The stream algebra can be extended to support a mix of ordered and unordered streams. Also fast packing algorithms that reduce the number of null records but do not eliminate them entirely might be considered.

The graph generated by the compiler and further transformed by my algorithm is not optimal. Several optimizations should be developed and applied to transform the graph further into a semantically equivalent, but more efficient form.

Bibliography

- [1] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conf. Proc.*, pages 483–485, 1967.
- [2] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schrooder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.
- [3] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *SIGGRAPH*, 2004.
- [4] Nathan A. Carr, Jesse D. Hall, and John C. Hart. GPU algorithms for radiosity and subsurface scattering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 51–59. Eurographics Association, 2003.
- [5] Eric Chan, Ren Ng, Pradeep Sen, Kekoa Proudfoot, and Pat Hanrahan. Efficient Partitioning of Fragment Shaders for Multipass Rendering on Programmable Graphics Hardware. *Graphics Hardware*, pages 69–78, 2002.
- [6] Randall J. Fisher and Henry G. Dietz. Compiling for SIMD within a register. In

- Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, pages 290–304. Springer-Verlag, 1999.
- [7] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a Stream-Processing Framework for Interactive rendering on Clusters. pages 693–702, 2002.
- [8] Kai Hwang. *Advance Computer Architecture*. McGraw-Hill, Inc, 1993.
- [9] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucec Khailany. Efficient conditional operations for data-parallel architectures. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 159–170. ACM Press, 2000.
- [10] Jens Kruger and Rudiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.
- [11] Michael Macedonia. The GPU enters computing’s mainstream. *IEEE Computer*, October 2003.
- [12] Michael McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh*. AK Peters Ltd., 2004.
- [13] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader Algebra. *SIGGRAPH*, 2004.
- [14] Michael D. McCool, Tiberiu Popa, and Kevin Moule. Stream GPU architectures. In *Technical Report CS-2003-23, School of Computer Science, University of Waterloo, August 2003*.

- [15] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68. Eurographics Association, 2002.
- [16] Kevin Moule and Michael D. McCool. Efficient Bounded Adaptive Tessellation of Displacement Maps. *Graphics Interface*, pages 171–180, 2002.
- [17] Stuart Oberman, Greg Favor, and Fred Weber. AMD 3DNow! technology: Architecture and implementations. *IEEE Micro*, 19(2):37–48, 1999.
- [18] John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon Rendering on a Stream Architecture. *Graphics Hardware*, pages 23–32, 2000.
- [19] John D. Owens, Brucek Khailany, Brian Towles, and William J. Dally. Comparing Reyes and OpenGL on a stream architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 47–56. Eurographics Association, 2002.
- [20] Alex Peleg and Uri Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42–50, 1996.
- [21] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 703–712. ACM Press, 2002.
- [22] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing

- streaming SIMD extensions on the pentium III processor. *IEEE Micro*, 20(4):47–57, 2000.
- [23] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucec Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 3–13. IEEE Computer Society Press, 1998.
- [24] Dezso Sima, Terence Fountain, and Peter Kacsuk. *Advanced Computer Architectures*. Addison-Wesley, 1997.
- [25] Allan Snavely, Greg Chun, Henri Casanova, Rob F. Van der Wijngaart, and Michael A. Frumkin. Benchmarks for grid computing: a review of ongoing efforts and future directions. *SIGMETRICS Perform. Eval. Rev.*, 30(4):27–32, 2003.
- [26] Chris J. Thompson, Sahngyun Hahn, and Mark Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317. IEEE Computer Society Press, 2002.