User Guide

AMD Stream Computing

A u g u s t 2 0 0 8

PRELIMINARY

Revision 1.1

AMD
Smarter Choice

Advanced Micro Devices, Inc.
One AMD Place
P.O. Box 3453
Sunnyvale, CA 94088-3453
www.amd.com

Preface

About This Document

This document provides a basic description of the AMD Stream Computing environment and components. It describes the basic architecture of stream processors and provides useful performance tips. This document also provides a guide for programmers who want to use the AMD Stream SDK to accelerate their applications.

Audience

This document is intended for programmers. Programming guides for Brook+ and CAL are provided. It assumes prior experience in writing code for CPUs and basic understanding of threads. While a basic understanding of GPU architectures is useful, this document does not assume prior graphics knowledge.

Contact Information

To submit questions or comments concerning this document, contact our technical documentation staff at: streamcomputing@amd.com.

For questions concerning AMD Stream products, please email: streamcomputing@amd.com.

For questions about developing with AMD Stream, please email: streamdeveloper@amd.com.

You can learn more about AMD Stream at: http://ati.amd.com/technology/streamcomputing.

We also have a growing community of AMD Stream users! Come visit us at the AMD Stream Developer Forum to find out what applications other users are trying on their AMD Stream products!

http://forums.amd.com/devforum/categories.cfm?catid=328

Organization

This document begins with an overview of the AMD Stream Computing programming models, the stream processor hardware description, and a discussion of performance and optimization when programming for stream processors. Chapters 2 and 3 are programming guides for the Brook+ language and CAL platform, respectively. Chapters 4 and 5 are the specifications for the Brook+ language and the CAL platform, respectively. The last section of this book is a glossary of acronyms and terms.

Conventions

The following conventions are used in this document.

`mono-spaced font`	A filename, file path, or code.
*	Any number of alphanumeric characters in the name of a code format, parameter, or instruction.
< >	Angle brackets denote streams.
[1,2)	A range that includes the left-most value (in this case, 1) but excludes the right-most value (in this case, 2).
[1,2]	A range that includes both the left-most and right-most values (in this case, 1 and 2).
{x | y}	One of the multiple options listed. In this case, x or y.
0.0	A single-precision (32-bit) floating-point value.
1011b	A binary value, in this example a 4-bit value.
7:4	A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first.
italicized word or phrase	The first use of a term or concept basic to the understanding of stream computing.

Related Documents

- AMD, *R600 Technology, R600 Instruction Set Architecture*, Sunnyvale, CA, est. pub. date 2007. This document includes the RV670 GPU instruction details.
- ISO/IEC 9899:TC2 - *International Standard - Programming Languages - C*
- Kernighan Brian W., and Ritchie, Dennis M., *The C Programming Language*, Prentice-Hall, Inc., Upper Saddle River, NJ, 1978.
- I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, *"Brook for GPUs: stream computing on graphics hardware,"* ACM Trans. Graph., vol. 23, no. 3, pp. 777–786, 2004.
- *AMD Intermediate Language (IL) Compiler Reference Manual*. Published by AMD.
- *CAL Image. AMD's Compute Abstraction Layer Program Binary Format Specification*. Published by AMD.
- Buck, Ian; Foley, Tim; Horn, Daniel; Sugerman, Jeremy; Hanrahan, Pat; Houston, Mike; Fatahalian, Kayvon. "BrookGPU" http://graphics.stanford.edu/projects/brookgpu/
- Buck, Ian. "Brook Spec v0.2". October 31, 2003. http://merrimac.stanford.edu/brook/brookspec-05-20-03.pdf

- *OpenGL Programming Guide*, at http://www.glprogramming.com/red/
- *Microsoft DirectX Reference Website*, at http://msdn.microsoft.com/archive/default.asp?

  url=/archive/en-us/directx9_c_Summer_04/directx/graphics/reference/reference.asp
- *GPGPU*: http://www.gpgpu.org, and Stanford BrookGPU discussion forum http://www.gpgpu.org/forums/

Contents

Figures

Chapter 1
AMD Stream Computing Overview

AMD Stream Computing harnesses the tremendous processing power of GPUs (stream processors) for high-performance, data-parallel computing in a wide range of applications.[1] The following is an overview of the AMD Stream Computing programming model, hardware, and performance.

1.1 The AMD Stream Computing Programming Model

The AMD Stream Computing Model includes a software stack and the AMD stream processors. Figure 1.1 illustrates the relationship of the AMD Stream Computing components.

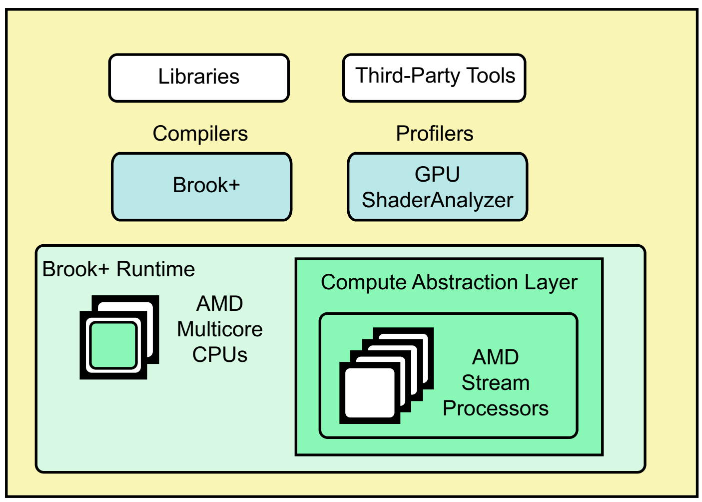

Figure 1.1 AMD Stream Software Ecosystem

The AMD Stream Computing software stack provides end-users and developers with a complete, flexible suite of tools to leverage the processing power in AMD stream processors. AMD software embraces open-systems, open-platform standards. The AMD open platform strategy enables AMD technology partners to develop and provide third-party development tools.

The software includes the following components:

- Compilers – like the Brook+ compiler with extensions for AMD devices.[2]

1. A *stream* is a collection of data elements of the same type that can be operated on in parallel.
2. See Chapter 2, “Brook+ Programming,” for using Brook+.

- Device Driver for stream processors – AMD Compute Abstraction Layer (CAL).[1]
- Performance Profiling Tools – GPU ShaderAnalyzer.
- Performance Libraries – ACML for optimized domain-specific algorithms.

The latest generation of AMD stream processors are programmed using the unified shader programming model. Programmable stream cores execute various user-developed programs, called *stream kernels* (or simply: kernels). These stream cores can execute non-graphics functions using a virtualized SIMD programming model operating on streams of data. In this programming model, known as *stream computing*, arrays of input data elements stored in memory are mapped onto a number of SIMD engines, which execute kernels to generate one or more outputs that are written back to output arrays in memory.

Each instance of a kernel running on a SIMD engine's thread processor is called a *thread*. A specified rectangular region of the output buffer to which threads are mapped is known as the *domain of execution*.

The stream processor schedules the array of threads onto a group of *thread processors*, until all threads have been processed. Subsequent kernels can then be executed, until the application completes. A simplified view of the AMD Stream Computing programming model and the mapping of threads to thread processors is shown in Figure 1.2 (also see Figure 1.9).

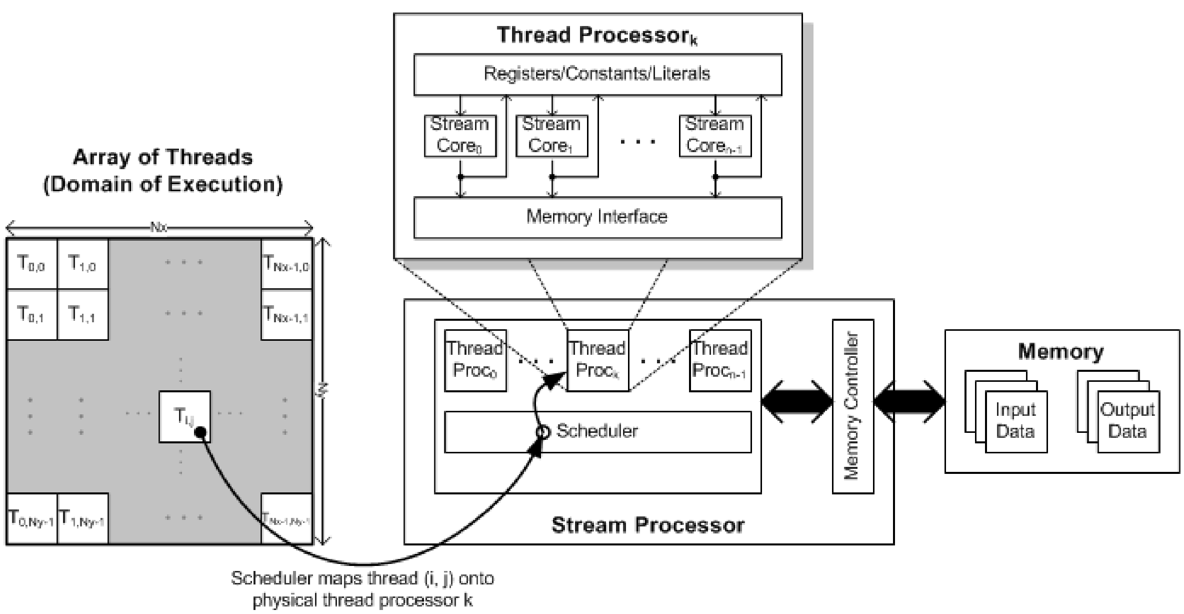

Figure 1.2 Simplified AMD Stream Computing Programming Model

1. When using CAL, it might not be necessary to use Brook+; instead, it is possible to use AMD IL. See Chapter 3, "AMD Compute Abstraction Layer (CAL) Programming Guide."

1.1.1 Pseudo Code Explanation of AMD Stream Computing

Another way to explain the AMD Stream Computing programming model is through pseudo code.

Matrix Sum – The following example sums two matrices.

The CPU code is:

```
void sum(float A[], float B[], float C[])
{
   for(int i=0; i<n; i++)
   {
      for(int j=0; j<m; j++)
      {
         float a0 = A[i][j];
         float b0 = B[i][j];

         C[i][j] = a0 + b0;
      }
   }
}
```

This code can be rewritten as to emphasize the data parallel operations:

```
float sum_kernel(int y, int x, float M0[], float M1[])
{
   float a0 = M0[y][x];
   float b0 = M1[y][x];

   return a0 + b0;
}

void sum(float A[], float B[], float C[])
{
   for(int i=0; i<n; i++)
   {
      for(int j=0; j<m; j++)
      {
         C[i][j] = sum_kernel(i, j, A, B);
      }
   }
}
```

The CPU executes the code serially such that `C[0][0]` is calculated before `C[0][1]`. However, the elements of `C` can be calculated independently of each other in any order. On a multi-CPU-core processor, they can also be calculated in parallel.

A multi-threaded version of the code might look like this:

```
void sum(float A[], float B[], float C[])
{
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<m; j++)
        {
            launch_thread{ C[i][j] = sum_kernel(i, j, A, B); }
        }
    }

    sync_threads{}
}
```

Effectively, this is the AMD Stream Computing programming model. The function `sum_kernel` is the kernel written by the developer. The array `C` is the output stream and defines the domain of execution (*n* x *m*). Independent threads that run `sum_kernel` execute and write at every location in `C`. The hardware takes the place of the nested for-loop.

Figure 1.3 illustrates the process of a matrix sum execution in a stream processor. Since the stream processor can operate in parallel with the CPU, `sync_threads` is used to wait for the threads to complete before continuing. The CPU can perform other tasks while the stream processor is processing.

High-level languages for AMD Stream Computing, such as Brook+, abstract the hardware details; no additional knowledge of stream processor hardware is required. The developer writes kernels to be executed on the stream processor, provides inputs and outputs, and defines the domains of execution.

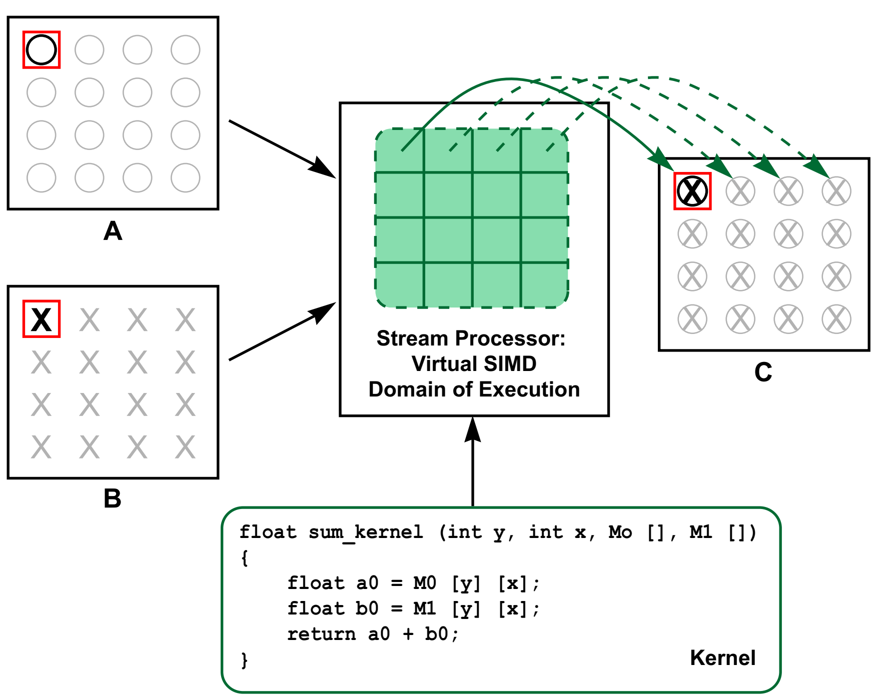

Figure 1.3 Stream Processor Execution

Matrix Multiply – This example multiplies two matrices (see Figure 1.4). This shows how some understanding of the hardware can improve performance.

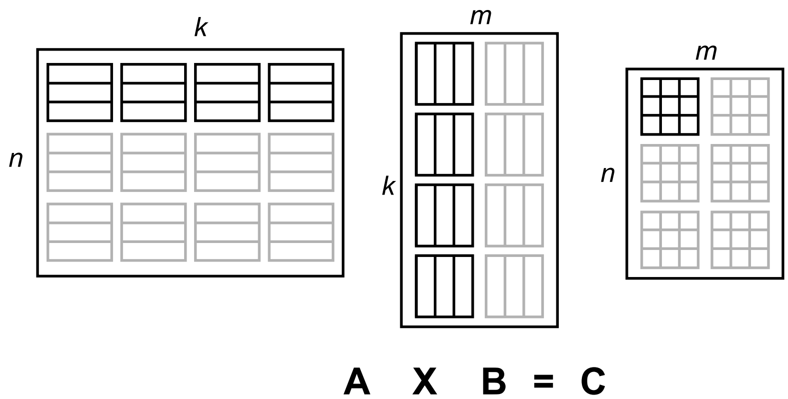

Figure 1.4 Matrix Multiply *(nxk) X (kxm)*

The CPU code is:

```
void matmult(float A[], float B[], float C[])
{
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<m; j++)
        {
            float total = 0;
            for(int c=0; c<k; c++)
                total += A[i][c] * B[c][j];

            C[i][j] = total;
        }
    }
}
```

The kernel that can be executed on the stream processor is shown in bold. The outer two for-loops represent the stream processor executing the kernel on the domain of execution of array C.

Again, this code can be rewritten as to emphasize the data parallel operations:

```
float matmult_kernel(int y, int x, int k,
                     float M0[], float M1[])
{
   float total = 0;
   for(int c=0; c<k; c++)
   {
      total += M0[y][c] * M1[c][x];
   }

   return total;
}

void matmult(float A[], float B[], float C[])
{
   for(int i=0; i<n; i++)
   {
      for(int j=0; j<m; j++)
      {
         launch_thread{C[i][j] = matmult_kernel(i, j, k, A, B);}
      }
   }

   sync_threads{}
}
```

One feature of stream processors is that each thread processor can perform parallel operations. So far, the examples indicate scalar operations in the kernel. If the compiler detects parallelization within a kernel, it tries to optimize it. For example, a thread processor can execute multiple multiplies and adds simultaneously. To take advantage of the stream processor's ability to perform multiple operations at the same time, the user can explicitly code in vector operations.

The following implementation uses the `float4` data type. This causes the thread processors to execute four operations at the same time:

```
float4 matmult_kernel(  int y, int x, int k,
                        float4 M0[], float4 M1[])
{
    float4 total = 0;
    for(int c=0; c<k/4; c++)
    {
        total += M0[y][c] * M1[x][c];
    }

    return total;
}

void matmult(float4 A[], float4 B'[], float4 C[])
{
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<m/4; j++)
        {
            launch_thread{C[i][j] = matmult_kernel(j, i, k, A, B');}
        }
    }

    sync_threads{}
}
```

Several key changes in this code maximize performance. Since inputs and outputs are now `float4` instead of `float`, the domain of execution dimensions decrease to *(n x (m/4))*; fewer threads are executed by the stream processor.

Also, the addressing for one of the arrays in the kernel has changed. To support maximum usage of `float4` operations, the second matrix, `B`, must be transposed to `B'`. The inner loop also decreases by a factor of four. The developer must decide if the extra step of transposing the input data is worth the cost.

If the input matrices are small, the transposition cost might not be offset by the performance gain in the kernel. If the matrices are large, the time to perform the transpose might be offset by the optimized kernel and yield a performance gain. If the input matrix sizes are variable, two separate code paths might be required for optimal performance.

The following sections explain how the stream processor executes kernels. It also teaches the developer how to optimize code for execution on the stream processor.

1.1.2 Brook+ Open-Source Data-Parallel C Compiler

Brook+ provides an explicit data-parallel C compiler using extensions to the standard ANSI C programming language. The Brook+ computational model, called *streaming*, goes beyond traditional, sequential programming languages by providing:

- Data Parallelism – Brook+ provides an intuitive mechanism for specifying single-instruction multiple-data (SIMD) operations.
- Arithmetic Intensity – the Brook+ interface encourages development of efficient algorithms by minimizing global communication and maximizing localized computation on stream processors.

The two key elements in the Brook+ language are:

- Stream – A collection of data elements of the same type that can be operated on in parallel. Streams are notated in angle brackets.
- Kernel – A parallel function that operates on every element of a domain of execution. Kernels are specified using the kernel keyword.

The following code shows a Brook+ kernel that adds two input streams and stores the results in an output stream. The kernel performs an implicit loop over each element in the output stream.

```
kernel
void sum(float a<>, float b<>, out float c<>)
{
   c = a + b;
}
```

As shown in Figure 1.5, the Brook+ software consists of:

- brcc – a source-to-source meta-compiler that translates Brook+ programs (`.br` files) into device-dependent kernels embedded in valid C++ source code. The generated C++ source includes the CPU code and the stream processor device code, both of which are later linked into the executable.
- brt – a runtime library that executes a kernel invoked from the CPU code in the application. Brook+ includes various runtimes for CPUs and stream processors; you can select the execution model at application run-time. The CPU runtime serves as a good debugging tool when developing stream kernels.

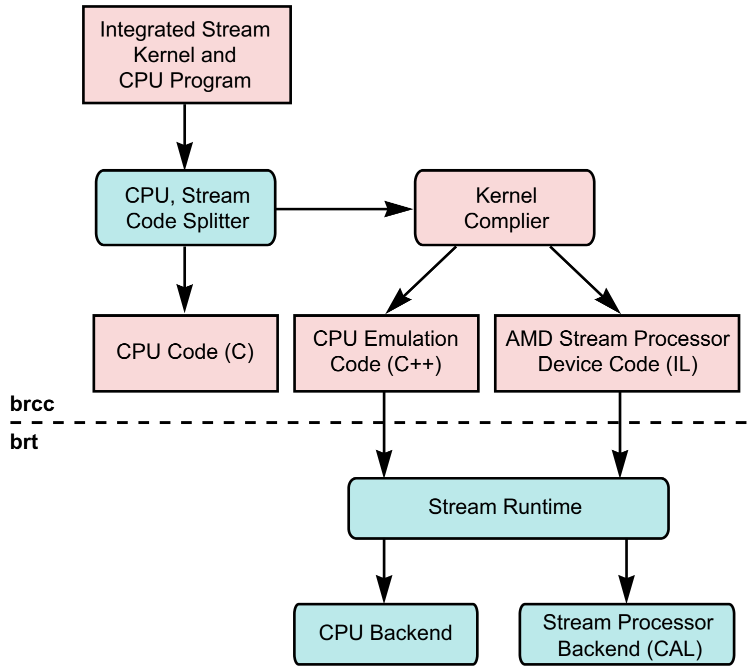

Figure 1.5 Brook+ Language Elements

AMD has enhanced brcc to produce the virtual ISA, called the AMD IL (for intermediate language). AMD also has enhanced the brt with a backend optimized for AMD stream processors using the CAL driver (see Section 1.1.3, "AMD Compute Abstraction Layer (CAL)," page 1-9).

1.1.3 AMD Compute Abstraction Layer (CAL)

The AMD Compute Abstraction Layer (CAL) is a device driver library that provides a forward-compatible interface to AMD stream processors (see Figure 1.6). CAL lets software developers interact with the stream processor cores at the lowest-level for optimized performance, while maintaining forward compatibility. CAL provides:

- Device Specific Code Generation
- Device Management
- Resource Management
- Kernel Loading and Execution
- Multi-device support
- Interoperability with 3D Graphics APIs

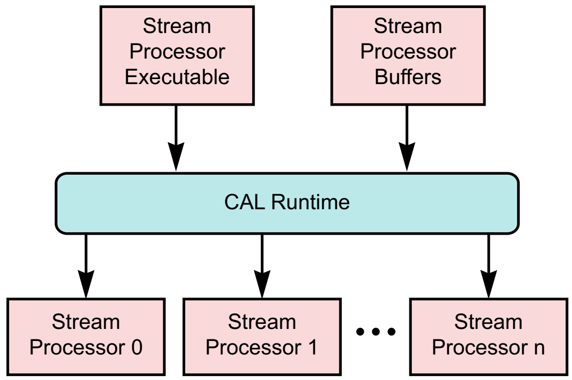

Figure 1.6 CAL Functionality

CAL includes a set of C routines and data types that allow higher-level software tools to control hardware memory buffers (device-level streams) and stream processor programs (device-level kernels). The CAL runtime accepts kernels written in AMD IL and generates optimized code for the target architecture. It also provides access to device-specific features.

1.1.4 GPU ShaderAnalyzer

The GPU ShaderAnalyzer is a performance-profiling tool developers can use to develop and profile stream kernels. It can be downloaded for free from the AMD developer web pages,
http://ati.amd.com/technology/streamcomputing/sdkdownload.html.

Features provided by the GPU ShaderAnalyzer include:

- Quick syntax checking of programs written in Brook+.
- Online kernel compilation to generate the equivalent AMD IL and the processor-specific ISA assembly. The generated assembly can be modified manually and used in a CAL application.
- Performance characterization of arithmetic, memory, and flow-control instructions.

The GPU ShaderAnalyzer has a simple graphical user interface. Figure 1.7 shows an example kernel, that was written in Brook+ and is converted to AMD IL. The generated AMD IL can be sent to the CAL runtime compiler for object code generation and subsequent execution.

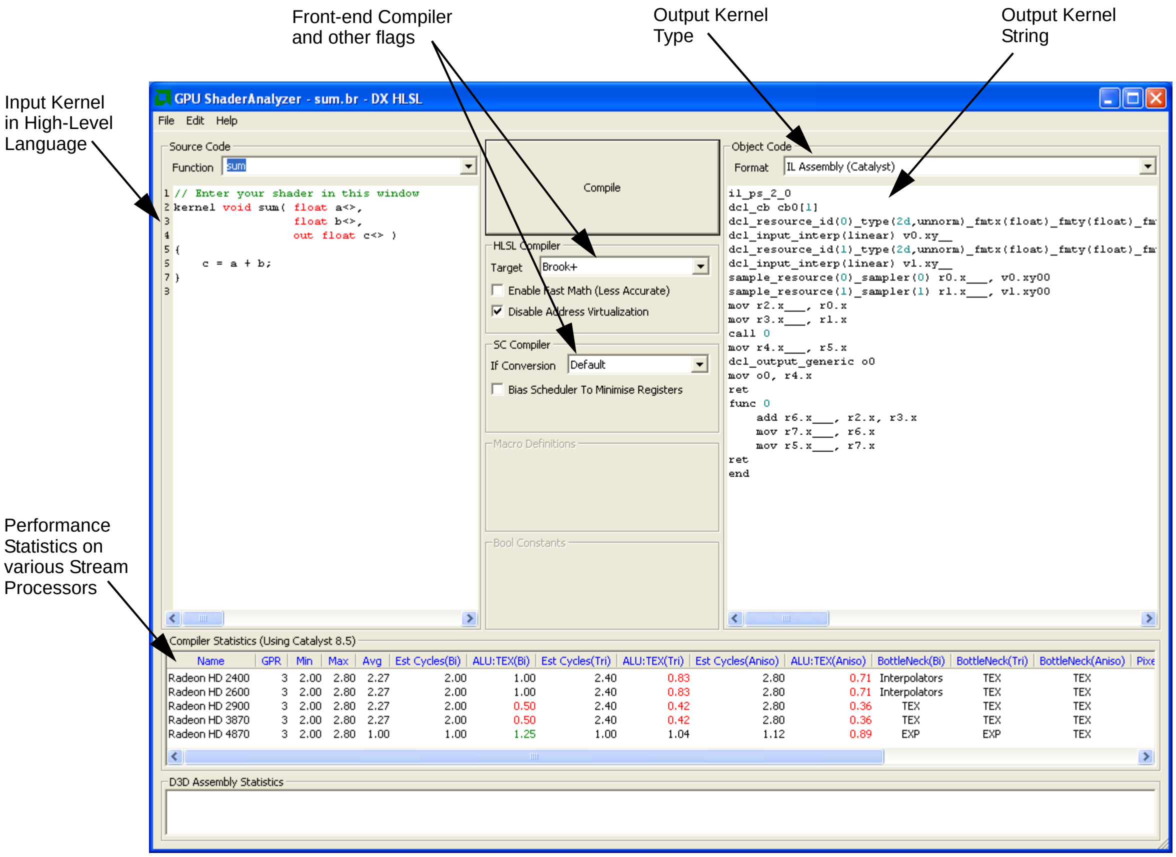

Figure 1.7 GSA User Interface Example

Note that:

- The input program can be edited directly in the Source Code window on the top-left.
- The function name must be the name of the Brook+ kernel.
- The target compiler must be set to Brook+ in the HLSL Compiler section.
- The output program type can be set using the Format selection tab in the Object Code section.

1.1.5 AMD Core Math Library (ACML)

The ACML includes a collection of commonly used mathematical software routines. It is optimized for AMD platforms and provides a quick path to high-performance development.

The ACML includes implementations of:

- Full Basic Linear Algebra Subroutines (BLAS)
- Linear Algebra Package (LAPACK) routines
- Fast Fourier Transform (FFT) routines

- Math transcendental routines
- Random Number Generator routines

The ACML includes a stream processing backend for load balancing of computations between the CPU and stream processor depending upon the suitability of the task for a particular architecture.[1] This is done at runtime.

1.2 Stream Processor Hardware Functionality

Figure 1.8 shows a simplified block diagram of a generalized stream processor.

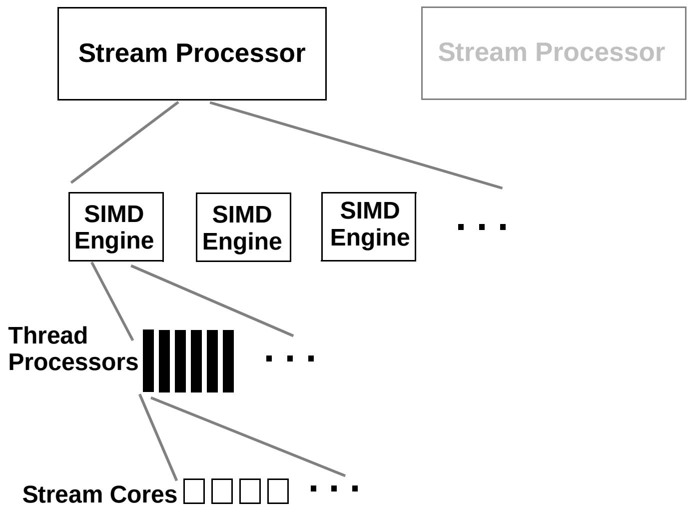

Figure 1.8 Generalized Stream Processor Structure

1.2.1 The Stream Processor

Figure 1.9 is a simplified diagram of an AMD stream processor. Different stream processors have different characteristics (such as the number of SIMD engines), but follow a similar design pattern.

Stream processors comprise groups of SIMD engines (see Figure 1.2). Each SIMD engine contains numerous thread processors, which are responsible for executing kernels, each operating on an independent data stream. Thread processors, in turn, contain numerous stream cores, which are the fundamental, programmable computational units, responsible for performing integer, single, precision floating point, double precision floating point, and transcendental operations. All thread processors within a SIMD engine execute the same instruction sequence; different SIMD engines can execute different instructions.

1. The stream-accelerated version of the ACML is called ACML-GPU. The ACML-GPU uses the stream processor to accelerate ACML routines that can benefit from stream acceleration. The ACML-GPU currently provides stream-accelerated implementations of SGEMM and DGEMM.

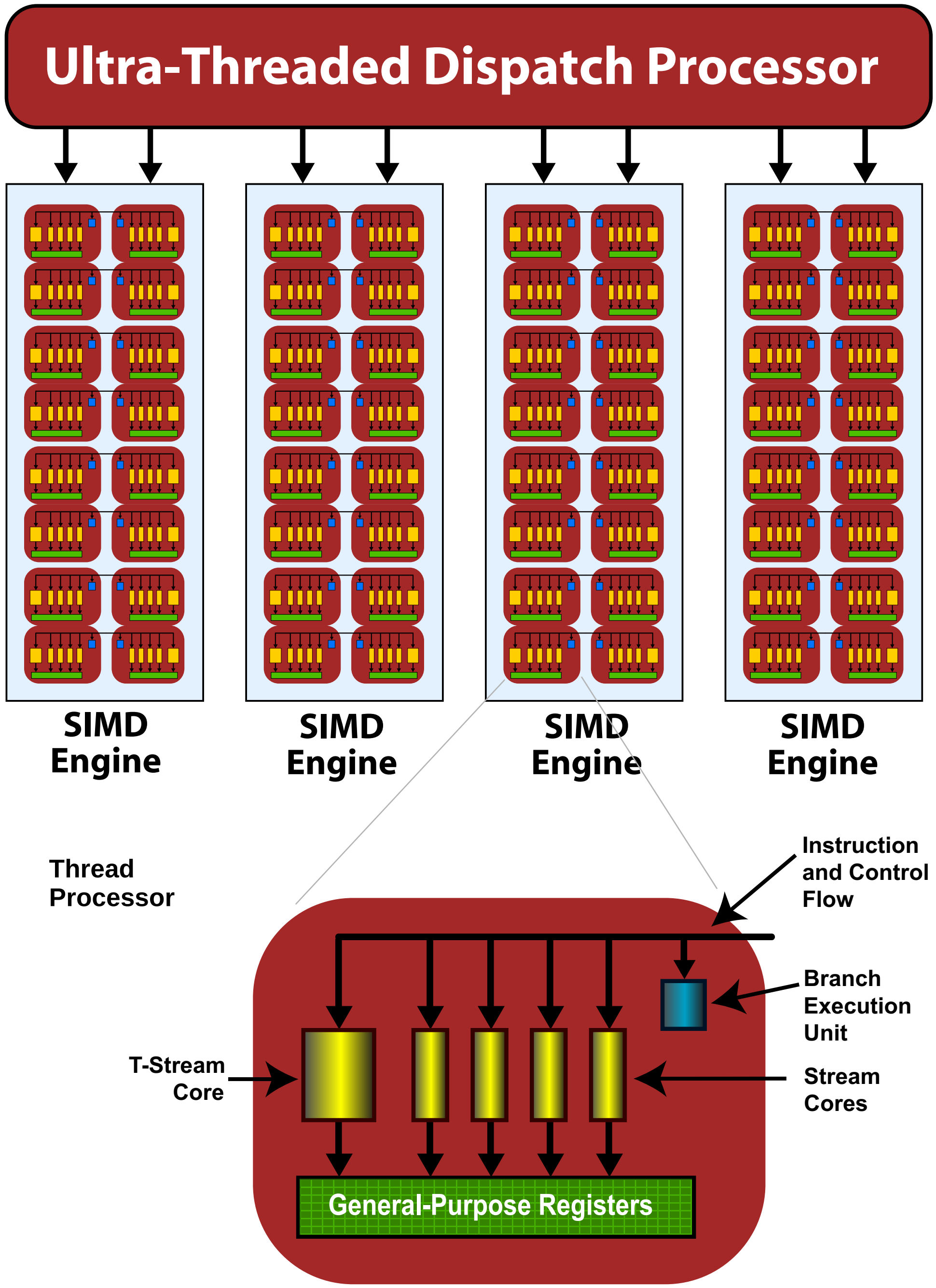

Figure 1.9 Simplified Block Diagram of the Stream Processor[1]

A thread processor is arranged as a five-way VLIW processor (see bottom of Figure 1.9). Up to five scalar operations can be co-issued in a very long instruction word (VLIW) instruction. Stream cores can execute single-precision floating point or integer operations. One of the five stream cores also can handle transcendental operations (sine, cosine, logarithm, etc.)[2]. Double-precision floating point operations are processed by connecting four of the stream cores

1. As described later, much of this is transparent to the programmer.

(excluding the transcendental core) to perform a single double-precision operation. The thread processor also contains one branch execution unit to handle branch instructions.

Different stream processors have different numbers of stream cores. For example, the ATI Radeon™ 3870 GPU (RV670) stream processor has four SIMD engines, each with 16 thread processors, and each thread processor contains five stream cores; this yields 320 physical stream cores.

1.2.2 Thread Processing

All thread processors within a SIMD engine execute the same instruction for each cycle. To hide latencies due to memory accesses and stream core operations, multiple threads are interleaved; thus, in a thread processor, up to four threads can issue four VLIW instructions over four cycles. For example, on the ATI Radeon™ 3870 GPU (RV670) stream processor, the 16 thread processors execute the same instructions, with each thread processor processing four threads at a time; effectively, this appears as a 64-wide SIMD engine. The group of threads that are executed together is called a *wavefront*.

The size of wavefronts can differ on different stream processors. For example, the ATI Radeon™ HD 2600 and the ATI Radeon™ HD 2400 graphics cards each have fewer thread processors in each SIMD engine on their stream processors compared to the ATI Radeon™ 3870 GPU (RV670) stream processor; therefore, the wavefront sizes are 32 and 16 threads, respectively. The AMD FireStream™ 9170 stream processor, which uses the RV670 stream processor, has a wavefront size of 64 threads.

SIMD engines operate independently of each other, so it is possible for each array to execute different instructions.

1.2.3 Flow Control

Flow control, such as branching, is done by combining all necessary paths as a wavefront. If threads within a wavefront diverge, all paths are executed serially. For example, if a thread contains a branch with two paths, the wavefront first executes one path, then the second path. The total time to execute the branch is the sum of each path time. An important point is that even if only one thread in a wavefront diverges, the rest of the threads in the wavefront execute the branch. The number of threads that must be executed during a branch is called the *branch granularity*. On AMD hardware, the branch granularity is the same as the wavefront granularity.

Example 1: If two branches, A and B, take the same amount of time *t* to execute over a wavefront, the total time of execution, if any thread diverges, is *2t*.

Loops execute in a similar fashion, where the wavefront occupies a SIMD engine as long as there is at least one thread in the wavefront still being processed.

2. For the actual operations, see the AMD *Compute Abstraction Layer (CAL) Technology Intermediate Language (IL) Reference Manual*.

Thus, the total execution time for the wavefront is determined by the thread with the longest execution time.

Example 2: If *t* is the time it takes to execute a single iteration of a loop; and within a wavefront all threads execute the loop one time, except for a single thread that executes the loop 100 times, the time it takes to execute that entire wavefront is *100t*.

1.2.4 Thread Creation

Wavefronts are composed of *quads*, which are groups of 2x2 threads in the domain. Quads are processed together. If there are non-active threads within a quad, the thread processors that would have been mapped to those threads are idle. The simplest example is a domain of execution of height or width one. In this case, since quads are not fully covered, the hardware is only half used because half the quad is empty.

Wavefront construction and order of thread execution are determined by the *rasterization order* of the domain of execution (see Figure 1.10). *Rasterization* is the process of mapping threads from the domain of execution to SIMD engines[1].

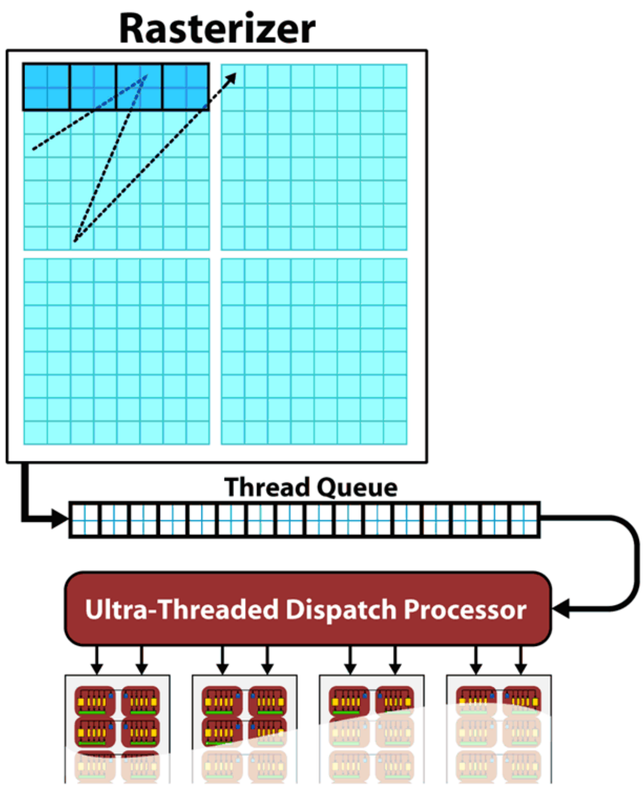

Figure 1.10 Rasterization of Threads to SIMD Engines

1. Rasterization is a carryover from graphics terminology, where it refers to the process of turning geometry, such as triangles, into pixels.

1.2.4.1 Rasterization

Rasterization follows a pre-set zig-zag-like pattern across the domain of execution. The exact pattern normally is not disclosed because it might change in subsequent stream processor generations. The pattern is based on multiples of 8x8 blocks (16 quads) within the domain, matching the size of a wavefront. For example, if the domain of execution is 16x16, the first 8x8 block maps to one wavefront and is executed in one SIMD engine. A second 8x8 block maps to another wavefront and is executed in another SIMD engine. This continues until all 8x8 blocks in the domain are mapped to SIMD engines.

1.2.4.2 Thread Optimization

AMD hardware is designed to maximize the number of active threads in a wavefront. So, if there are partial 8x8 blocks, the stream processor tries to fill the rest of the wavefront from other blocks, but within the quad limitation. For example, if the domain is of height 2, the wavefront is constructed using blocks of height 2 and width 32. Thus, having domains that are a multiple of 8x8 is not necessary, but might be more efficient.

This rasterization process is transparent to the user, but can affect memory access performance, as described in Section 1.2.5, “Accessing Memory,” page 1-16.

1.2.5 Accessing Memory

Accessing stream processor local memory typically is an order of magnitude faster than accessing remote (system or CPU) memory. However, stream cores (see Figure 1.8) do not directly access memory; instead, they issue memory requests through dedicated hardware units. When a thread tries to access memory, the thread is transferred to the appropriate fetch unit. The thread is then deactivated until the access unit finishes accessing memory. Meanwhile, other threads can be active within the SIMD engine, contributing to better performance. The data fetch units handle three basic types of memory operations: loads, stores, and streaming stores. Stream processors now can store writes to random memory locations using global buffers.

1.2.5.1 Global Buffer

The global buffer lets applications read from, and write to, arbitrary locations in input buffers and output buffers, respectively. When using a global buffer, memory-read and memory-write operations from the stream kernel are done using regular stream processor instructions with the global buffer used as the source or destination for the instruction. The programming interface is similar to load/store operations used with CPU programs, where the relative address in the read/write buffer is specified.

1.2.5.2 Memory Loads

Memory loads are done by addressing the desired location in the input memory using the fetch unit. The fetch units can process either 1D or 2D addresses.

These addresses can be *normalized* or *un-normalized*. Normalized coordinates are between 0.0 and 1.0 (inclusive). For the fetch units to handle 2D addresses and normalized coordinates, pre-allocated memory segments must be bound to the fetch unit so that the correct memory address can be computed. For a single kernel invocation, up to 128 memory segments can be bound at once. The maximum number of 2D addresses is 8192x8192. When accessing a global buffer, of which only one can be bound at a time, addresses must be un-normalized, 1D coordinates. Memory loads are usually cached, except for loads from a global buffer, which are not cached.

1.2.5.3 Memory Stores

When using a global buffer, each thread can write to an arbitrary location within the global buffer. Only one global buffer is allowed to be bound at a time for a particular kernel invocation. The same global buffer must be used for loads and stores. Global buffers use a linear memory layout. If consecutive addresses are written, the SIMD engine issues a burst write for more efficient memory access.

1.2.5.4 Streaming Stores

Kernels can perform streaming writes in up to eight separate memory segments. The streaming writes occur only once per kernel invocation: only one write is allowed per segment, and the write location is implicitly computed based on each thread's location in the domain of execution. For example, the thread at location <1,1> in the domain would write to location <1,1> in each bound memory segment. For these addresses to computed implicitly, the sizes of the bound memory segments must be the same and specified beforehand.

1.2.5.5 Memory Tiling

There are many possible physical memory layouts for data streams. AMD stream processors can access memory in a tiled or in a linear arrangement.

- Linear – A linear layout format arranges the data linearly in memory such that element addresses are sequential. This is the layout that is familiar to CPU programmers. This format must be used for global buffers.
- Tiled – A tiled layout format has a pre-defined sequence of element blocks arranged in sequential memory addresses (see Figure 1.11). Translating from user address space to the tiled arrangement is transparent to the user. Tiled memory layouts provide an optimized memory access pattern to make more efficient use of the RAM attached to the stream processor. This contributes to lower latency.

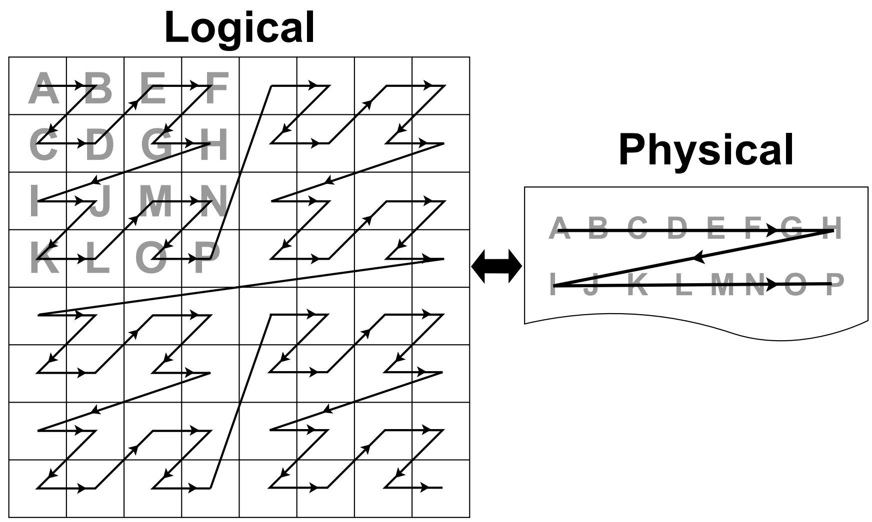

Figure 1.11 One Example of a Tiled Layout Format

1.2.6 Host-to-Stream Processor Communication

The following subsections discuss the communication between the host (CPU) and the stream processor. This includes an overview of the PCI Express® bus, processing API calls, and DMA transfers.

1.2.6.1 PCI Express Bus

Communication and data transfers between the system and the stream processor occur on the PCI Express® (PCIe®) channel. AMD Stream Computing cards use PCIe 2.0 x16 (second generation, 16 lanes). Generation 1 x16 has a theoretical maximum throughput of 4 GBps in each direction. Generation 2 x16 doubles the throughput to 8 GBps in each direction. Actual transfer performance is CPU and chipset dependent.

Transfers from the system to the stream processor are done either by the *command processor* or by the *DMA engine*. The stream processor also can read and write system memory directly from the SIMD engine through kernel instructions over the PCIe® bus.

1.2.6.2 Processing API Calls: The Command Processor

The host application does not interact with the stream processor directly. A driver layer translates and issues commands to the hardware on behalf of the application.

Most commands to the stream processor are buffered in a command queue on the host side. The command queue is flushed to the stream processor, and the commands are processed by it, only when a kernel program is executed. Flushing sends the current state of the command queue to the stream processor. There is no guarantee as to when commands from the command queue are executed, only that they are executed in order. Unless the stream processor is busy, commands are executed immediately.

Command queue elements include:

- Kernel execution calls
- Kernels
- Constants

1.2.6.3 DMA Transfers

Direct Memory Access (DMA) memory transfers can be executed separately from the command queue using the DMA engine on the stream processor. DMA calls are executed immediately; and the order of DMA calls and command queue flushes is guaranteed.

DMA transfers can occur asynchronously. This means that a DMA transfer is executed concurrently with other system or stream processor operations. However, data is not guaranteed to be ready until the DMA engine signals that the event or transfer is completed. The application can query the hardware for DMA event completion. If used carefully, DMA transfers are another source of parallelization.

The thread processors handle non-DMA memory transfers.

1.2.7 Stream Processor Scheduling

Stream processors are very efficient at running large numbers of threads in a manner transparent to the application. Each stream processor uses the large number of threads to hide memory access latencies by having the resource scheduler switch the active thread in a given thread processor whenever the current thread is waiting for a memory access to complete. This time multiplexing is also used to hide the latency of stream core operations resulting from pipelining. Hiding memory access latencies requires that each thread contain a large number of calculations.

Figure 1.12 shows the timing of a simplified execution of threads in a single thread processor. At time 0, the threads are queued and waiting for execution. In this example, only four threads (T0…T3) are scheduled for the processor. The hardware limit for the number of active threads is dependent on the resource usage (such as the number of active registers used) of the program being executed. An optimally programmed stream processor typically has thousands of active threads.

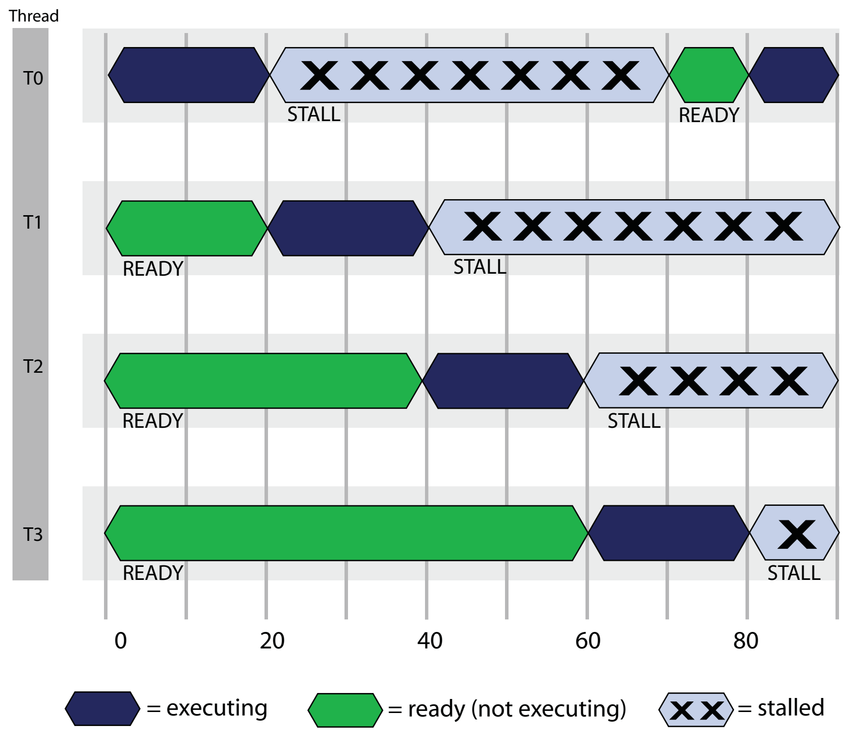

Figure 1.12 Simplified Execution Of Threads On A Single Thread Processor

At runtime, thread T0 executes until cycle 20; at this time a stall occurs due to a memory fetch request. The scheduler then begins execution of the next thread, T1. Thread T1 executes until it stalls or completes. New threads execute, and the process continues until the available number of active threads is reached. The scheduler then returns to the first thread, T0.

If the data thread T0 is waiting for has returned from memory, T0 continues execution. In the example in Figure 1.12, the data is ready, so T0 continues. Since there were enough threads and stream core operations to cover the long memory latencies, the thread processor does not idle. This method of memory latency hiding helps the stream processor achieve maximum performance.

If the data for thread T0 is not ready, the thread processor waits until thread T0 is ready to execute, even if there are other threads ready to execute, as demonstrated in Figure 1.13.

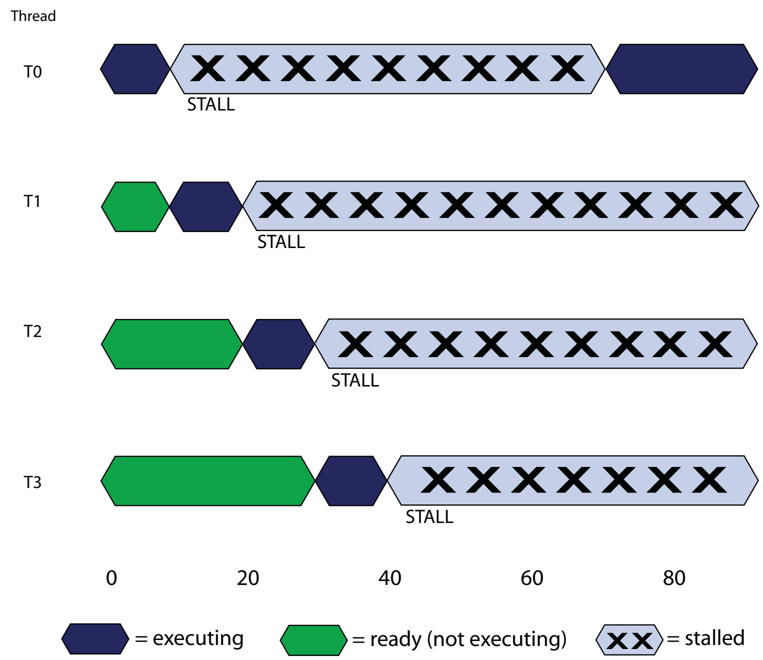

Figure 1.13 Thread Processor Stall Due to Data Dependency

The causes for this situation are discussed in the following sections.

1.3 Performance

This section discusses performance and optimization when programming for stream processors.

1.3.1 Analyzing Stream Processor Kernels

Kernels must be compiled to native hardware instructions. The AMD GPU ShaderAnalyzer (Figure 1.14) can provide the instruction set architecture (ISA) disassembly. This tool can show the instructions executed on the hardware, as well as the number of active registers used.

Looking at the ISA of an example program (see Figure 1.14), instructions are grouped into *clauses*. A clause is a set of sequential instructions that executes without *pre-emption*. There are three types of instructions: stream core, local memory fetch, and memory read/write. Clauses can only contain one type of instruction. Only one clause is loaded onto a SIMD engine or the local memory fetch units at a time; however, multiple clauses can be executed in parallel because each SIMD can run a different clause.

Figure 1.14 shows an implementation of matrix multiply using Brook+. The resulting ISA code contains eight clauses (00…07). Of these, 00, 02, 03, and 05 are stream core clauses; 01 and 06 are branch clauses; 04 is a fetch clause; and 07 is a memory write clause. There are seven stream core instructions and two fetch instructions.

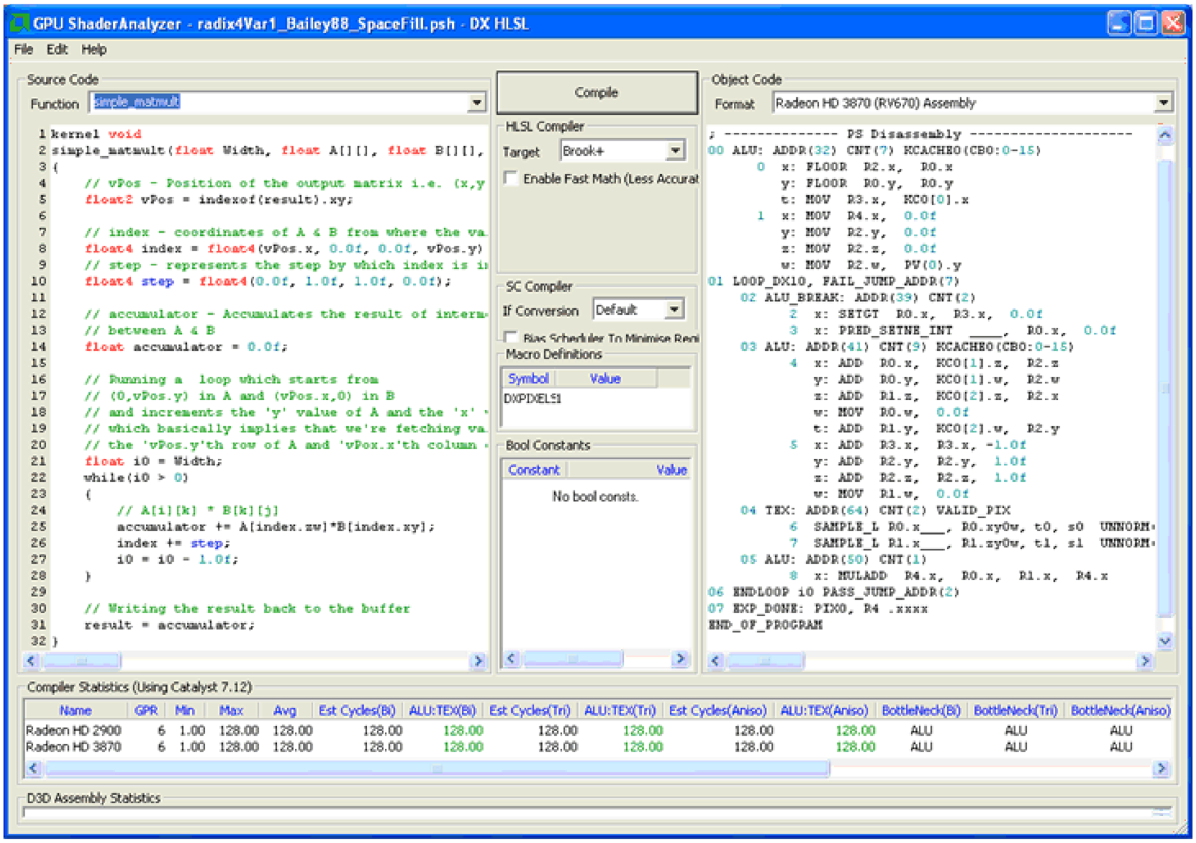

Figure 1.14 AMD GPU ShaderAnalyzer Output

1.3.2 Estimating Performance

Estimating the theoretical performance of a kernel running on a stream processor is important because it helps developers identify and remove performance bottlenecks.

The last section shows the components of the instructions of a kernel. This is needed for the theoretical estimates. The other information needed consists of:

- Number of stream cores
- Number of local memory fetch units
- Memory bus size
- Engine clock frequency
- Memory clock frequency

For the ATI Radeon™ 3870 GPU (RV670) stream processor, the number of thread processors that execute the VLIW instructions is 64. The memory bus size is 256 bits. The engine and memory clocks are dependent on the stream processor (see the technical specifications for a specific stream processor for the rates). A typical ATI Radeon™ HD 3870 graphics card, which uses the RV670 stream processor, has an engine clock of 775 MHz and a memory clock of 1125 MHz.

A kernel with only stream core instructions has a theoretical performance of:

$$\frac{\text{(\# threads) x (\# VLIW stream core instructions/thread)}}{\text{(stream core instructions / clk) x (3D engine clock)}}$$

The number of threads is the size of the domain of execution. Taking the ATI Radeon™ 3870 GPU (RV670) stream processor as an example, a one stream core instruction kernel with a domain of two million threads theoretically executes in:

$$\frac{\text{(2M threads) x (1 stream core instruction/thread)}}{\text{(64 stream core instructions / clk) x 775 MHz}} = 0.04 \text{ ms}$$

A kernel with only a single fetch instruction has a theoretical performance of:

$$\frac{\text{(\# threads) x (\# fetch instructions/thread)}}{\text{(fetches / clk) x (3D engine clock)}} = \frac{\text{2M x 1}}{\text{16 x 775 MHz}}$$
$$= 0.16 \text{ ms}$$

Local memory fetch units operate on the engine clock; thus, the 3D engine speed was used in the calculation above.

Memory performance estimation is based on the total amount of data being read from, and written to, memory per thread:

$$\frac{\text{(\# threads) x (in + out bits per thread)}}{\text{(bus) x (memory clock)}}$$

A simple copy kernel (one byte in and one byte out) with a domain of two million threads has a theoretical memory performance of:

$$\frac{\text{(2M threads) x (16 bits)}}{\text{(256 bits) x (1125 MHz x 2DDR)}} = 0.056 \text{ ms}$$

All hardware units run in parallel. Thus, the theoretical performance is the worst case of the three estimates. In the example of a kernel with one stream core instruction, one fetch instruction, and one byte input and output, the theoretical runtime would be 0.16 ms. This kernel is considered fetch-bound because the local memory fetch units are the bottleneck.

Note that the theoretical performance serves only as a guide. As kernel complexity increases, the ability to model the hardware becomes more difficult. Also, the above memory performance model is based on ideal (sequential) memory access patterns. Section 1.3.3, “Additional Performance Factors” explores additional factors which affect performance.

1.3.3 Additional Performance Factors

This section describes potential factors that can impact kernel performance on the stream processor.

1.3.3.1 Register Usage

The number of active wavefronts depends on the active register usage of a kernel. This can be determined from the ISA disassembly provided by the GPU ShaderAnalyzer or other tools. Compilers try to optimize for the best register use; however, manual optimizations often can yield better results. Optimizing register counts yields performance gains through better memory latency hiding. However, a stream-core-bound kernel is bound by the peak stream core performance, even if many threads are active simultaneously.

When too many active registers are used, the stream processor places excess registers into memory. If this happens, performance is significantly impacted.

1.3.3.2 Domain Size

Stream processors have deep pipelines and many parallel execution units. Thus, stream processors require a large number of threads to be executed for maximum efficiency. This, however, is highly application workload dependent.

As mentioned in Section 1.2.2, "Thread Processing," page 1-14, and Section 1.2.4, "Thread Creation," page 1-15, threads are executed on the hardware in wavefronts and quads. It is recommended that, at a minimum, domains have a height or width of a multiple of two.

1.3.3.3 Stream Core to Fetch Instruction Ratio

One often-cited kernel statistic is the stream core-to-fetch (instructions) ratio. As shown in Section 1.3.2, "Estimating Performance," page 1-22, there must be enough stream core instructions to hide the fetch latencies. This consideration is not intended for initially developing kernel programs, but rather for cases where the performance of the kernel program is not as expected. This ratio is device-specific.

1.3.3.4 Memory Fetch Instructions

Since there are normally significantly more stream core resources than memory fetch resources, it is important that the developer keep memory fetch instructions to a minimum. Every memory fetch instruction takes at least one cycle. If the kernel is designed to fetch from consecutive data locations, then vector fetches can make more efficient use of the fetch resources. For example, a kernel can issue a fetch for a `float4` type in one cycle versus four separate float fetches in four cycles. Sometimes, the compiler consolidates fetches; however, if there is math involved in calculating addresses, the compiler might not be able to perform the optimization for the developer. One solution is to explicitly load data into registers as a first step (prefetching), rather than calling for fetches in the code as needed.

1.3.3.5 Thread Processor Use

Most developers are used to programming with scalar operations. The compiler attempts to parallelize kernels into VLIW instructions for the developer. However, if instructions are highly dependent on each other, the VLIW might have low occupancy; then, the thread processors are under-used. One optimization is to vectorize not just fetches, but also threads. This is done by combining multiple threads into a single thread and writing out multiple results with a vector data type, such as `float4`.

Since threads can write out up to eight vector types, it is possible to do much more work per thread by vectorizing them. This not only minimizes the number of stream core operations, but also might reduce the number of memory fetches.

Further optimization is achieved by having data ready in registers, since reading from registers is faster than fetching data from the cache.

1.3.3.6 Memory Access Patterns

The hardware is optimized for sequential memory access within, and between, threads. This is due to the way the DRAM and the cache are set up. On a memory fetch, an entire cache line is returned, which accelerates the next fetch in the sequence. Also, tiled memory works with thread rasterization (discussed in Section 1.2.4, “Thread Creation,” page 1-15) to accelerate memory fetches and increase performance. This is because consecutively created threads are likely to have their fetches in the cache already, leading to less stalling in the thread processor.

When a stream is formatted with a linear layout, performance can be negatively affected. More cache lines might be fetched to service the reads than from a tiled format.

Random accesses into memory, and fetch patterns that consistently access the same memory bank and channel (all fetches going to the same physical memory chip), cause the greatest degradation in memory performance.

Since memory access patterns can throw off performance estimates, it is possible to isolate the stream core and fetch performance by reducing input stream sizes to just one element. This determines if a kernel is memory bound or not, since by reducing the input stream size, the input stream data remains in the cache. This technique only works on fetches that do not depend on a value written from the kernel.

1.3.3.7 Command Processor

Since the command queue is flushed on every execution of a stream processor program, short kernels and small domains can cause many gaps to be inserted in the execution pipeline.

Having too large of a command queue also can affect performance. The buffer in the command processor has a finite size. Thus, very large command queues must be repackaged into smaller queues. As a result, extra overhead can occur when handling very large command queues.

1.3.3.8 Bus Transfers

Ideally, total stream processor time measures not only the kernel compute time, but also the transfer of data over the system bus between the host and the stream processor, or between multiple stream processors. Bus transfers are highly platform dependent, so running the application on another system sometimes can be the quickest attempt at optimization.

Another method for improving performance is to hide the data transfer time with other work. Since the stream processor can read and write data directly from host memory, for some applications it might be better to leave the input or output streams in host memory and avoid any explicit bus transfer steps.

Since DMA transfers are asynchronous, they can be hidden through other CPU or stream processor computations. This can be achieved by subdividing a large domain and transferring data for subsequent kernels during prior kernel executions. However, it is important to ensure that asynchronous transfers have completed before a kernel tries to use transferred data for computation.

Chapter 2
Brook+ Programming

This chapter is for developers using the Brook+ language to develop applications for AMD stream processors. See *Brook+ Language Specification* for a complete development guide and language specification. Also, see Section 3.1, "Introduction," page 3-1, for an introduction to the stream processor architecture.

2.1 Prerequisites

The following subsections detail the system requirements, the installation procedure, and the steps for enabling syntax highlighting in Visual Studio.

2.1.1 System Requirements

Installing Brook+ SDK (compiler, runtime, and sample applications) on your system requires:

- Windows XP 32-bit, XP 64-bit (both SP2).
- Vista 32-bit, Vista 64-bit (both SP1).
- All ATI Radeon HD 2000+ series, ATI Radeon HD 3870, ATI Radeon HD 4850, ATI Radeon HD 4870, ATI FireGL V7700, and AMD FireStream 9170 graphics cards.
- Visual Studio 2005 (VS8) and 2008 (VS9) – Brook+ platform and samples are in VS8.
- Linux (both 32-bit and 64-bit): Red Hat Enterprise Linux 5.1, SUSE Linux Enterprise Server (SLES) 10 SP1.

2.1.2 Installation

Access http://ati.amd.com/technology/streamcomputing/sdkdwnld.html, and download the following files:

- ATI Catalyst™ Driver - Brook+ and CAL require ATI Catalyst version 8.1 or above. Choose the version of ATI Catalyst that matches your operating system.
- AMD Stream SDK v1.2-beta (Win XP 32) or AMD Stream SDK v1.2-beta (Win XP 64) - Choose the one that matches your operating system. The executable file installs the Brook+ compiler and runtime, as well as the AMD CAL software.

- GPU ShaderAnalyzer (optional) - This is a tool for analyzing the performance of pixel and vertex shaders on ATI Radeon graphics cards. It gives you accurate performance estimates for shaders and lets you view disassembly of the generated hardware shader. It can be used as a GUI tool for interactive tuning of shaders, or, in command line mode, to generate detailed reports. It supports DirectX9, DirectX10 & OpenGL.
- ACML (optional) - The AMD Core Math Library.

The AMD Stream SDK compressed file contains a setup executable that installs the Brook+ SDK and the CAL SDK. This package contains the following files:

File and location	Description
`<BROOKROOT>\`	Install directory
`doc\`	Documentation
`platform\`	
`brook.sln`	Visual Studio solution
`brcc\src\`	Brook+ Compiler source
`include\`	Brook+ include files
`runtime\src\`	Runtime source
`utils\`	Misc. make files and tools
`samples\`	Brook+ sample apps
`sdk\`	SDK tree
`bin\`	
`brcc.exe`	Release build of the Brook+ compiler
`brcc_d.exe`	Debug build of the Brook+ compiler
`include\`	Brook+ include files
`lib\`	
`brook.lib`	Release build of the Brook+ runtime library
`brook_d.lib`	Debug build of the Brook+ runtime library
`utils\`	
`vs8_syntaxHighlighting\`	Brook+ syntax highlighting files

To install the files:

1. Install AMD CAL (Compute Abstraction Layer) using the CAL installer, and follow the prompts.
   a. The installer adds the location of the CAL `dlls` to the path. The default location is:

      `C:\Program Files\AMD\<CAL SDK>\lib\xp32`

   b. The installer adds the environment variable `CALROOT` and sets it to the CAL install path. The default value is:

      `CALROOT = C:\Program Files\AMD\<CAL SDK>`

2. Install Brook+ using the provided installer, and follow the prompts.
   a. The installer adds the environment variable `BROOKROOT` and sets it to the Brook+ install path. The default value is:

      `BROOKROOT = C:\Program Files\AMD\<BROOK+ SDK>\`

2.1.3 Syntax Highlighting in Visual Studio

To enable syntax highlighting for *.br* files in Visual Studio:

1. If not already present, copy `usertype.dat` in `<BROOKROOT>\utils\vs8_syntaxHighlighting\` to `<Visual Studio 8>\Common7\IDE\`.
   Otherwise, append the contents of `usertype.dat` onto `<Visual Studio 8>\Common7\IDE\usertype.dat`.
2. Open Visual Studio 8, and select Tools→Options. Expand Text Editor and select File Extension. Enter *br* in the Extension box, choose your favorite editor, and click Add then OK to close window.
3. Close Visual Studio 8; then, restart this program.

See the associated `readme` file (located in `vs8_syntaxHighlighting\`) for more information.

2.2 A Sample Application

Brook+ comes in two components: the compiler (`brcc.exe`) and the Brook+ runtime libraries. Building an application consists of

1. Using the Brook+ compiler to compile the Brook+ source code into a C++ file. This contains the CPU and stream processor code.
2. Compiling the C++ file with the rest of the application and link it with the Brook+ runtime libraries.

The following subsections detail writing, building, executing, debugging, and logging a sample application.

2.2.1 Writing

The following is an example Brook+ source code for `sum.br` that adds two streams and outputs to a third. Brook+ source files normally are given a `.br` extension.

```
Sum.br

#include <stdio.h>

kernel void sum(float a<>, float b<>, out float c<>)
{
    c = a + b;
}
```

```
int main(int argc, char** argv)
{
   int i, j;
   float a<10, 10>;
   float b<10, 10>;
   float c<10, 10>;

   float input_a[10][10];
   float input_b[10][10];
   float input_c[10][10];

   for(i=0; i<10; i++)
   {
      for(j=0; j<10; j++)
      {
         input_a[i][j] = (float) i;
         input_b[i][j] = (float) j;
      }
   }

   streamRead(a, input_a);
   streamRead(b, input_b);

   sum(a, b, c);

   streamWrite(c, input_c);

   for(i=0; i<10; i++)
   {
      for(j=0; j<10; j++)
      {
         printf("%6.2f ", input_c[i][j]);
      }
      printf("\n");
   }

   return 0;
}
```

Brook+ code is very similar to C/C++. Note the following limitations.

First, `brcc` functions like a C compiler; thus, programs must adhere to standard C constructions (for example: variables are declared at the beginning of code blocks). The Brook+ compiler has no built-in preprocessor. If the kernel code uses preprocessor directives, the Brook+ file must be processed by a preprocessor before it is passed to the Brook+ compiler. The Brook+ compiler reports a problem when there is a preprocessor directive inside the kernel code, but passes preprocessor directives in non-kernel code to the C++ compiler invoked in the second step of the compilation.

For more complex applications, carefully partition the C code and the Brook+ code into manageable, easily maintainable sections. So, instead of using `main`, a function can be declared there and called from a C/C++ source file.

2.2.1.1 Kernels

From the example on page 2-3:

```
kernel void sum(float a<>, float b<>, out float c<>)
{
   c = a + b;
}

...

   sum(a, b, c);
```

Kernels are functions that run on the stream processor. The kernel is invoked on every element of the stream. Kernels are executed by calling them, just as in C with the actual parameters.

Kernels are written like C, but with some extensions and limitations (see the *Brook+ Language Specification* for a complete listing). In the following example, *a* and *b* are input streams, and *c* is the output stream. Streams use angle brackets. In this situation, the API automatically handles stream addressing.

2.2.1.2 Streams

From the example on page 2-3:

```
float a<10, 10>;
float b<10, 10>;
float c<10, 10>;
```

Streams are created using angle brackets (rather than square brackets used for arrays in C/C++). The hardware natively supports only 1D arrays up to 8192 elements, and 2D arrays up to 8192·8192 elements, where an element is the stream data type (for example: `float4`). Higher dimensions and larger sizes have limited support through address virtualization at compile time (possibly affecting the performance). For example, a 1D array can be virtualized to 64M (8192x8192) elements. See Section 2.2.2, "Building," page 2-6, for enabling address virtualization; also see Section 4.1 of the *Brook+ Language Specification* for more details.

2.2.1.3 Handling Streams

From the example on page 2-3:

```
streamRead(a, input_a);
streamRead(b, input_b);
...
streamWrite(c, input_c);
```

Streams cannot be accessed directly by the application. Data must be copied between streams and memory using `streamRead()` and `streamWrite()`.

`streamRead(stream *, void *);`	Copies data from memory to stream.
`streamWrite(stream *, void *);`	Copies data from stream to memory.

2.2.2 Building

Use the following steps to build:

Step 1. Compile with `brcc.exe`, which can be found in `<BROOKROOT>\sdk\bin\`

```
brcc [-ghklopr] [-o <prefix>] [-p shader] <.br file>
```

`-g`	`enable fast math (less accurate)`
`-h`	help (print this message)
`-k`	keep generated AMD IL program (in `foo.il`)
`-l`	insert #line directives into generated code
`-o prefix`	prefix prepended to all output files
`-p shader`	cpu or cal (can specify multiple)
`-r`	disable address virtualization

In the example on page 2-3, use:

```
brcc.exe -o sum sum.br
```

This compiles the Brook+ file `sum.br` and generates a C++ file, `sum.cpp`. (see Figure 2.1)

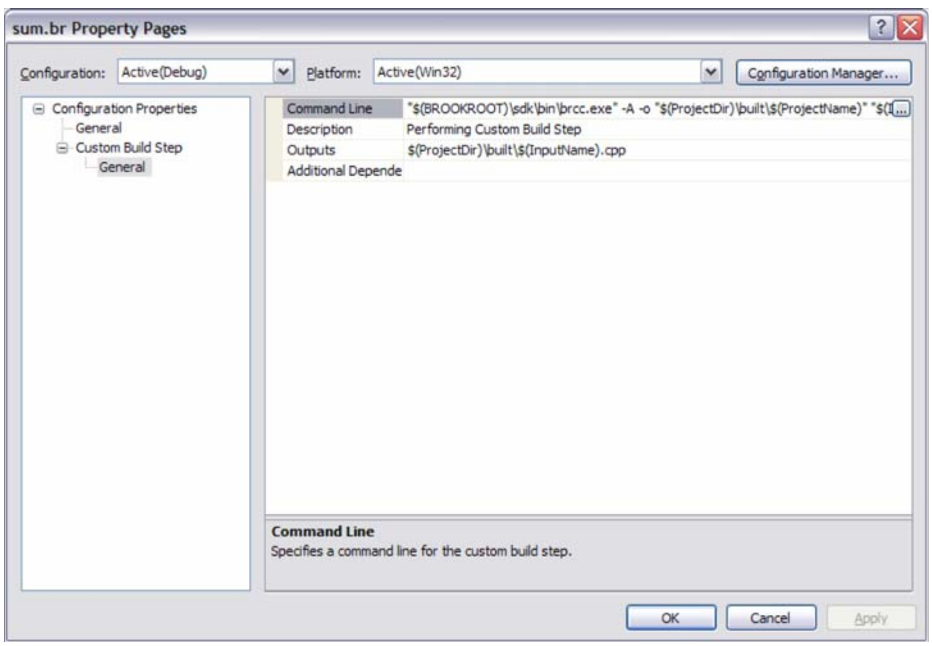

Figure 2.1 Compiling a Brook+ File and Generating a C++ File

In Visual Studio, you can add the Brook+ compilation step as a custom build event for the Brook+ file. Right-click on the Brook+ file in the project, and select Properties.

In *Command Line*, add the compiler command. For *Outputs*, add the location of the generated C++ file. You then can add the generated C++ file to the project. Subsequent Brook+ compiles overwrite the existing C++ file.

Brook+ header files are located in `<BROOKROOT>\sdk\include`.

3. Add `brook.lib` to Additional Linker dependencies. This library can be found in `<BROOKROOT>\sdk\lib\`.
4. Compile the application with the generated C++ files.

To use a `makefile`, see `<BROOKROOT>\samples\util\build` for examples.

2.2.3 Executing

If the installation was followed correctly and the build was successful, run the executable. If the application does not run, then at least one path has not been set.

2.2.4 Debugging

When debugging an application, debugging happens on the generated C++ source, not on the original Brook+ source. For a complete example, see Section 2.4, "Example of Generated C++ Code for sum.br," page 2-12.

There is no hardware debugging of stream kernels (for example: __sum_cal_desc); it is not possible to step through the kernel code. The kernel inputs and outputs can be inspected (before a `streamRead` and after a `streamWrite`). Kernels can be written so that intermediate data can be output to streams and inspected.

Alternatively, kernels can be stepped through and debugged as usual using the CPU emulation mode (for example: __sum_cpu and __sum_cpu_inner). To enable CPU emulation, create and set the environment variable:

```
BRT_RUNTIME = cpu
```

To return to the CAL backend, either delete the environment variable or set it to:

```
BRT_RUNTIME = cal
```

2.2.5 Logging

Brook+ can log various internal states to a log file. Use the following steps to enable this feature:

Step 1. Invoke the preprocessor macro `#define BROOK_LOGGER_ENABLED = 1` (currently, this is the default, set in `logger.hpp`).

Step 2. Add an environment variable `BRT_LOG_FILE` with the log file name.

Optionally, set the environment variable `BRT_LOG_MASK` to a decimal integer representing a bitmask of the types of information to log. The values corresponding to each log type are listed in `logger.hpp`. For example, to enable logging of function entries and warning messages, set BRT_LOG_MASK to 5.

Similarly, the compiler execution can be logged by setting the corresponding environment variables BRCC_LOG_FILE and BRCC_LOG_MASK. Note, however, that compiler logging coverage is incomplete.

Logging is enabled only in the debug versions of the tools. It is disabled in the release versions for performance reasons.

2.3 Included Samples

The Brook+ folder contains sample applications that can be built using the included makefiles or the included Visual Studio solution file <BROOKROOT>\samples\samples.sln.

Release builds of the samples are pre-built and located in: <BROOKROOT>\samples\bin\.

2.3.1 Simple Matrix Multiply Example

This example is a standard matrix multiply. The code presented here is excerpted from the `simple_matmult` example found in the `samples` directory.

```
////////////////////////////////////////////////////////////////////////!
//! C = A * B
//! \param Width The value for which the loop runs over the matrices
//! \param A Input matrix A(MxK)
//! \param B Input matrix B(KxN)
//! \param result Output matrix(MxN)
//!
///////////////////////////////////////////////////////////////////////
kernel void
simple_matmult(float Width, float A[][], float B[][], out float result<>)
{
    // vPos - Position of the output matrix i.e. (x,y)
    float2 vPos = indexof(result).xy;

    // index - coordinates of A & B from where the values are fetched
    float4 index = float4(vPos.x, 0.0f, 0.0f, vPos.y);
    // step - represents the step by which index is incremented
    float4 step = float4(0.0f, 1.0f, 1.0f, 0.0f);

    // accumulator - Accumulates the result of intermediate calculation
    // between A & B
    float accumulator = 0.0f;

    // Running a loop which starts from
    // (0,vPos.y) in A and (vPos.x,0) in B
    // and increments the 'y' value of A and the 'x' value of B
    // which basically implies that we're fetching values from
    // the 'vPos.y'th row of A and 'vPox.x'th column of B
    float i0 = Width;
    while(i0 > 0)
    {
        // A[i][k] * B[k][j]
        accumulator += A[index.zw]*B[index.xy];
        index += step;
        i0 = i0 - 1.0f;
    }
```

```
    // Writing the result back to the buffer
    result = accumulator;
}
int main(int argc, char** argv)
{
    float A<Height, Width>;
    float B<Width, Height>;
    float C<Height, Height>;
    float* inputA;
    float* inputB;
    float* output;
    …
    streamRead(A, inputA);
    streamRead(B, inputB);
    …
    simple_matmult((float)Width, A, B, C);
    …
    streamWrite(C, output);
    …
}
```

Starting at `main`, three streams are created representing the input (`A` and `B`) matrices and the output matrix (streams are used to represent a matrix). Then, three corresponding memory buffers are declared (`inputA`, `inputB`, and `inputC`).

Next, `streamRead()` copies data from `inputA` to stream `A`, and data from `inputB` to stream `B`.

The line `simple_matmult((float)Width, A, B, C);` binds the kernel to the size parameter `Width`, the input streams `A` and `B`, and the output stream `C`; this also triggers execution of the kernel by the stream processor. In a simple matrix multiply operation, the kernel reads in one row vector from one matrix and a column vector from another matrix; it applies a dot product to the two vectors, and writes out the result. In the example above, the kernel is invoked at each data location in the output stream. The kernel:

1. loops over the row of matrix A,
2. loops over the column of matrix B,
3. fetches a value from each matrix, and
4. accumulates the values.

A feature used by this kernel is vector data types (`float2` and `float4`). Brook+ can support data types of up to four elements. Elements can also be accessed in any combination. This is also known as *swizzling*.

There is also a difference between the stream inputs in this kernel compared to those of the earlier sum kernel. Here, the inputs are passed in using square brackets, which means that the input streams are treated as a memory array, and data elements are addressed directly. This is also known as a *gather stream*. An important distinction between kernel code and C code is that gather streams must be accessed using vector types, instead of multiple square brackets. For example, A[x][y] is not allowed.

To determine which row/column the kernel must access, the output location to which the kernel is writing must be specified. This is done through the `indexof()` function, which returns an integer (x,y) position of the output domain.

In the `while` loop, column values are read from matrix A and multiplied against values from matrix B. The `accumulator` variable accumulates the resulting values.

Like the earlier sum example, the result is written without bracket operators. Brook+ automatically writes the data out to the correct location; in this case, the location found in `indexof()` of the output stream.

2.3.2 Optimized Matrix Multiply Example

A disadvantage to the above kernel is that the same data is reused by the kernel at separate output locations. For example, at neighboring output locations, the kernel is reusing the same row vector or column vector data. Generally, fetching data from memory is expensive relative to processing data inside the stream processor.

One optimization technique is to perform more computations in the kernel, so that the reads are aggregated. This is the kernel from the optimized matrix multiply sample.

```
kernel void
optimized_matmult(float loopVar0,
       float4 A1[][], float4 A2[][], float4 A3[][], float4 A4[][],
       float4 A5[][], float4 A6[][], float4 A7[][], float4 A8[][],
       float4 B1[][], float4 B2[][], float4 B3[][], float4 B4[][],
       out float4 C1<>, out float4 C2<>, out float4 C3<>,
       out float4 C4<>, out float4 C5<>, out float4 C6<>,
       out float4 C7<>, out float4 C8<>)
{
   // vPos - Position of the output matrix i.e. (x,y)
   float2 vPos = indexof(C1).xy;

   // Setting four210
   float4 four210 = float4(4.0f, 2.0f, 1.0f, 0.0f);

   // index - coordinates of A & B from where the values are fetched
   float4 index = float4(vPos.x, vPos.y, four210.w, four210.w);

   // Declaring and initializing accumulators
   float4 accumulator1 = four210.wwww;
   float4 accumulator2 = four210.wwww;
   float4 accumulator3 = four210.wwww;
   float4 accumulator4 = four210.wwww;
   float4 accumulator5 = four210.wwww;
   float4 accumulator6 = four210.wwww;
   float4 accumulator7 = four210.wwww;
   float4 accumulator8 = four210.wwww;

   float i0 = loopVar0;

   while(i0 > 0.0f)
   {
      // Fetching values from A
      float4 A11 = A1[index.wy];
      float4 A22 = A2[index.wy];
```

```
        float4 A33 = A3[index.wy];
        float4 A44 = A4[index.wy];
        float4 A55 = A5[index.wy];
        float4 A66 = A6[index.wy];
        float4 A77 = A7[index.wy];
        float4 A88 = A8[index.wy];

    // Fetching values from B
        float4 B11 = B1[index.xw];
        float4 B22 = B2[index.xw];
        float4 B33 = B3[index.xw];
        float4 B44 = B4[index.xw];
        accumulator1 +=  A11.xxxx * B11.xyzw + A11.yyyy * B22.xyzw +
                         A11.zzzz * B33.xyzw + A11.wwww * B44.xyzw;
        accumulator2 +=  A22.xxxx * B11.xyzw + A22.yyyy * B22.xyzw +
                         A22.zzzz * B33.xyzw + A22.wwww * B44.xyzw;
        accumulator3 +=  A33.xxxx * B11.xyzw + A33.yyyy * B22.xyzw +
                         A33.zzzz * B33.xyzw + A33.wwww * B44.xyzw;
        accumulator4 +=  A44.xxxx * B11.xyzw + A44.yyyy * B22.xyzw +
                         A44.zzzz * B33.xyzw + A44.wwww * B44.xyzw;
        accumulator5 +=  A55.xxxx * B11.xyzw + A55.yyyy * B22.xyzw +
                         A55.zzzz * B33.xyzw + A55.wwww * B44.xyzw;
        accumulator6 +=  A66.xxxx * B11.xyzw + A66.yyyy * B22.xyzw +
                         A66.zzzz * B33.xyzw + A66.wwww * B44.xyzw;
        accumulator7 +=  A77.xxxx * B11.xyzw + A77.yyyy * B22.xyzw +
                         A77.zzzz * B33.xyzw + A77.wwww * B44.xyzw;
        accumulator8 +=  A88.xxxx * B11.xyzw + A88.yyyy * B22.xyzw +
                         A88.zzzz * B33.xyzw + A88.wwww * B44.xyzw;

        index += four210.wwwz;
        // Reducing iterator
        i0 = i0 - 1.0f;
    }

    C1 = accumulator1;
    C2 = accumulator2;
    C3 = accumulator3;
    C4 = accumulator4;
    C5 = accumulator5;
    C6 = accumulator6;
    C7 = accumulator7;
    C8 = accumulator8;
}
```

This example optimizes the kernel by:

- Using input streams of vector data types. In this case, `float4` is used so that every fetch retrieves four values simultaneously.
- Writing to eight streams simultaneously from the kernel. Using the CAL backend, Brook+ supports up to eight outputs per kernel. Each invocation of this kernel calculates 4×8 = 32 output values. Aggregating the memory fetches per kernel significantly increases the efficiency of the stream processor.
- Separating the two input matrices into multiple slices. This decreases the number of calculations needed to determine the addresses. The same

address used to fetch from different inputs representing the slices of the matrices.

Figure 2.2 illustrates the optimized matrix multiplication.

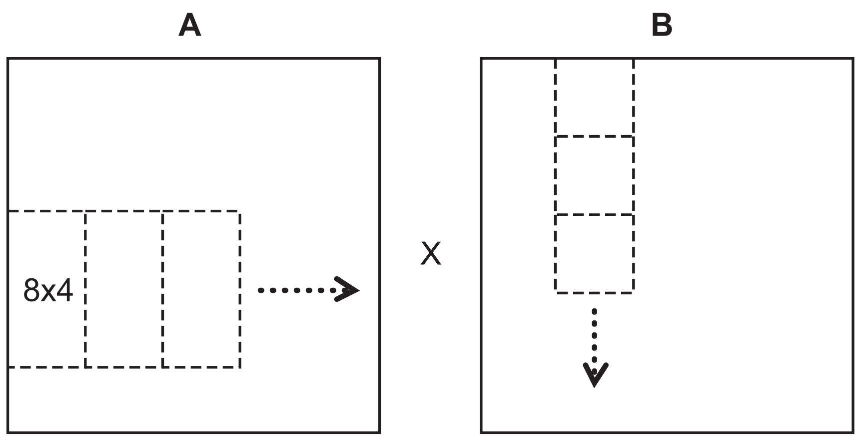

Figure 2.2 Optimized Matrix Multiplication

During each iteration of the loop in this kernel implementation, an 8x4 sub-matrix is fetched from matrix A, and a 4x4 sub-matrix is fetched from matrix B. Multiplying these two sub-matrices results in an 8x4 sub-matrix. In the next iteration of the loop, the next 8x4 sub-matrix in the row is fetched from A, and the next 4x4 sub-matrix in the column is fetched from B. These matrices are multiplied and accumulated with the earlier results. The resulting 8x4 matrix is output to a stream.

2.4 Example of Generated C++ Code for **`sum.br`**

```
////////////////////////////////////////////
// Generated by BRCC v0.1
// BRCC Compiled on: Nov  5 2007 16:24:44
////////////////////////////////////////////

#include <brook/brook.hpp>
#include <stdio.h>


namespace {
  using namespace ::brook::desc;
  static const gpu_kernel_desc __sum_cal_desc = gpu_kernel_desc()
    .technique( gpu_technique_desc()
        .pass( gpu_pass_desc(
            "il_ps_2_0\n"
            "dcl_cb cb0[1]\n"
            "dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)\n"
            "dcl_input_generic_interp(linear) v0.xy__\n"
            "dcl_resource_id(1)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)\n"
            "dcl_input_generic_interp(linear) v1.xy__\n"
            "sample_resource(0)_sampler(0) r0.x, v0.xy00\n"
            "sample_resource(1)_sampler(1) r1.x, v1.xy00\n"
            "mov r2.x, r0.xxxx\n"
            "mov r3.x, r1.xxxx\n"
            "call 0\n"
            "mov r4.x, r5.xxxx\n"
            "dcl_output_generic o0\n"
            "mov o0, r4.xxxx\n"
            "ret\n"
```

```
            "func 0\n"
            "add r6.x, r2.xxxx, r3.xxxx\n"
            "mov r7.x, r6.xxxx\n"
            "mov r5.x, r7.xxxx\n"
            "ret\n"
            "end\n"
            " \n"
            "##!!BRCC\n"
            "##narg:3\n"
            "##s:1:a\n"
            "##s:1:b\n"
            "##o:1:c\n"
            "##workspace:1024\n"
            "##!!multipleOutputInfo:0:1:\n"
            "##!!fullAddressTrans:0:\n"
            "##!!reductionFactor:0:\n"
            "")
            .sampler(1, 0)
            .sampler(2, 0)
            .interpolant(1, kStreamInterpolant_Position)
            .interpolant(2, kStreamInterpolant_Position)
            .output(3, 0)
      )
    );
  static const void* __sum_cal = &__sum_cal_desc;
}

static const char *__sum_ps30= NULL;
void  __sum_cpu_inner(const __BrtFloat1  &a,
                     const __BrtFloat1  &b,
                     __BrtFloat1  &c)
{
  c = a + b;
}
void  __sum_cpu(::brook::Kernel *__k, const std::vector<void *>&args)
{
  ::brook::StreamInterface *arg_a =
(::brook::StreamInterface *) args[0];
  ::brook::StreamInterface *arg_b =
(::brook::StreamInterface *) args[1];
  ::brook::StreamInterface *arg_c =
(::brook::StreamInterface *) args[2];

  do {
    Addressable <__BrtFloat1  > __out_arg_c((__BrtFloat1 *) __k->FetchElem(arg_c));

    __sum_cpu_inner (Addressable <__BrtFloat1 >((__BrtFloat1 *) __k->FetchElem(arg_a)),
Addressable <__BrtFloat1 >((__BrtFloat1 *) __k->FetchElem(arg_b)), __out_arg_c);

    *reinterpret_cast<__BrtFloat1 *>(__out_arg_c.address) =
__out_arg_c.castToArg(*reinterpret_cast<__BrtFloat1 *>
    (__out_arg_c.address));
  } while (__k->Continue());
}

void  sum (::brook::stream a,
        ::brook::stream b,
        ::brook::stream c) {
  static const void *__sum_fp[] = {
     "ps30", __sum_ps30,
     "cal", __sum_cal,
     "cpu", (void *) __sum_cpu,
     NULL, NULL };
  static ::brook::kernel  __k(__sum_fp);

  __k->PushStream(a);
  __k->PushStream(b);
  __k->PushOutput(c);
  __k->Map();

}


int  main(int  argc, char  **argv)
```

```
{
  int  i;
  int  j;
  ::brook::stream a(::brook::getStreamType(( float  *)0), 10 , 10,-1);
  ::brook::stream b(::brook::getStreamType(( float  *)0), 10 , 10,-1);
  ::brook::stream c(::brook::getStreamType(( float  *)0), 10 , 10,-1);
  float  input_a[10][10];
  float  input_b[10][10];
  float  input_c[10][10];

  for (i = 0; i < 10; i++)
  {
    for (j = 0; j < 10; j++)
    {
      input_a[i][j] = (float ) (i);
      input_b[i][j] = (float ) (j);
    }

  }

  streamRead(a, input_a);
  streamRead(b, input_b);
  sum(a, b, c);
  streamWrite(c, input_c);
  for (i = 0; i < 10; i++)
  {
    for (j = 0; j < 10; j++)
    {
      printf("%6.2f ", input_c[i][j]);
    }

    printf("\n");
  }

  return 0;
}
```

2.5 Building Brook+

Both the release and debug builds of the Brook+ compiler and runtime libraries come pre-built; however, they also can be built using the provided source.

The path to the pre-built SDK (binary, library, and headers) is:

<BROOKROOT>\sdk\

2.5.1 Visual Studio

You can build the brcc and the Brook+ runtime using the included Visual Studio solution file, which is located at:

<BROOKROOT>\platform\brook.sln

The configuration for getting the Debug or the Release executable is available through the Configuration pull-down menu.

The default output directories of builds using Visual Studio are:

brcc.exe: <BROOKROOT>\platform\brcc\bin\xp_x86_32
brook.lib: <BROOKROOT>\platform\runtime\lib\xp_x86_32

Files in the SDK tree are not replaced with the new builds. If make is installed, in <BROOKROOT>\platform:

- run make updatesdk to copy the debug to the SDK tree,
- or run make udpatesdk RELEASE=1 to copy the release builds to the SDK tree.

2.5.2 Command Line

The Brook+ tools can be built from the command line or through a Cygwin shell.

1. Install the entire Cygwin tools suite (see http://www.cygwin.com). Ensure that make is installed.
2. If not installed in the previous step, install flex and bison. These are used to generate files for the compiler. These generated files have already been included in the installation.
3. The Visual Studio compiler (cl.exe) and linker (link.exe) must be in the path. Default location is:

   `C:\Program Files\Microsoft Visual Studio 8\VC\bin`

   Note that in the path, the Visual Studio link.exe must come before the Cygwin link.exe.
4. Run *make* at <BROOKROOT>\platform\ for a debug build and run *make RELEASE=1* for a release build.

Unlike the Visual Studio builds, the SDK tree is rebuilt and overwritten with the new Brook+ builds.

To clean the build, use make clean for debug builds and make clean RELEASE=1 for release builds.

Chapter 3
AMD Compute Abstraction Layer (CAL) Programming Guide

3.1 Introduction

The AMD Compute Abstraction Layer (CAL) provides an easy-to-use, forward-compatible interface to the high-performance, floating-point, parallel processor arrays found in AMD stream processors. CAL, part of AMD's Stream Computing Software stack (see Figure 1.1), abstracts the hardware details of the AMD stream processor. It provides the following features:

- Device management
- Resource management
- Code generation
- Kernel loading and execution

CAL provides a device driver library that allows applications to interact with the stream cores at the lowest-level for optimized performance, while maintaining forward compatibility.

> Note: For developers beginning to develop stream computing software for stream processors, AMD recommends becoming familiar with the basic concepts of stream processor programming by looking at the Brook+ software. (Brook+ is a higher-level language that is easier to use, but does not provide all the functionality that CAL does.)

Brook+ provides an easy-to-use, high-level interface for stream computing, including a CAL-based runtime backend that is optimized for AMD stream processors. The CAL API is ideal for performance-sensitive developers because it minimizes software overhead and provides full-control over hardware-specific features that might not be available with higher-level tools.

The following subsections provide an overview of the CAL system architecture, stream processor architecture, and the execution model that it provides to the application.

3.1.1 CAL System Architecture

A typical CAL application includes two parts:

- a program running on the host CPU (written in C/C++), the *application*, and
- a program running on the stream processor, the *kernel* (written in a high-level language, such as AMD IL).

The CAL API comprises one or more stream processors connected to one or more CPUs by a high-speed bus. The CPU runs the CAL and controls the stream processor by sending commands using the CAL API. The stream processor runs the kernel specified by the application. The stream processor device driver program (CAL) runs on the host CPU.

Figure 3.1 is a block diagram of the various CAL system components and their interaction. Both the CPU and stream processor are in close proximity to their local memory subsystems. In this figure:

- Local memory subsystem – the CAL local memory. This is the memory subsystem attached to each stream processor. (From the perspective of CAL, the Stream Processor is local, and the CPU is remote.)
- System memory – the single memory subsystem attached to all CPUs.

CPUs can read from, and write to, the system memory directly; however, stream processors can read from, and write to:

- their own local stream processor memory using their fast memory interconnects, as well as
- system memory using PCIe.

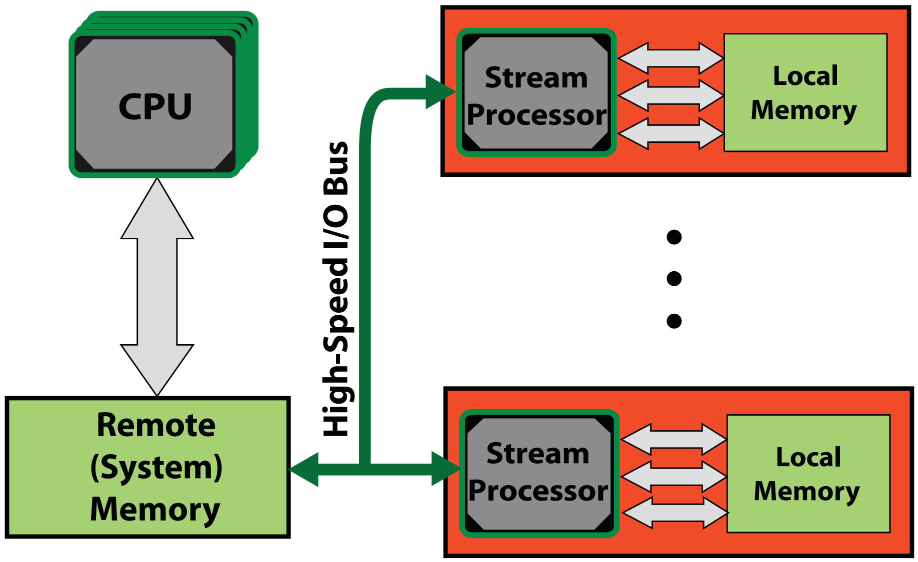

Figure 3.1 CAL System Architecture

The CAL runtime allows managing multiple stream processors directly from the host application. This lets applications divide computational tasks among multiple parallel execution units and scale the application in terms of computational performance and available resources. With CAL, applications control the task of partitioning the problem and scheduling among different stream processors (see Section 3.7, "Advanced Topics.")

3.1.1.1 CAL Device

The CAL API exposes the stream processors as a Single Instruction, Multiple Data (SIMD) array of computational processors. These processors execute the loaded kernel. The kernel reads the input data from one or more *input resources*, performs computations, and writes the results to one or more *output resources* (see Figure 3.2). The parallel computation is invoked by setting up one or more outputs and specifying a domain of execution for this output. The device has a scheduler that distributes the workload to the SIMD processors.

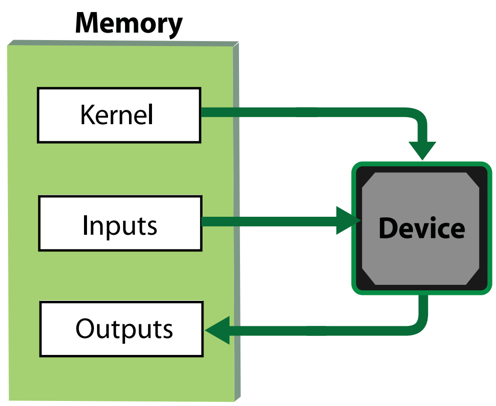

Figure 3.2 CAL Device and Memory

Since the stream processor can access both local device memory and remote memory, inputs and outputs to the kernel can reside in either memory subsystem. Data can be moved across different memory systems by the CPU, stream processor, or the DMA engine. Additional inputs to the kernel, such as constants, can be specified. Constants typically are transferred from remote memory to local memory before the kernel is invoked on the device.

3.1.1.2 Stream Processor Architecture

The AMD stream processor has a parallel micro-architecture for computer graphics and general-purpose parallel computing applications. Any data-intensive application that can be mapped to one or more kernels and the input/output resource can run on the AMD stream processor.

Figure 3.3 shows a block diagram of the AMD stream processor and other components of a CAL application.

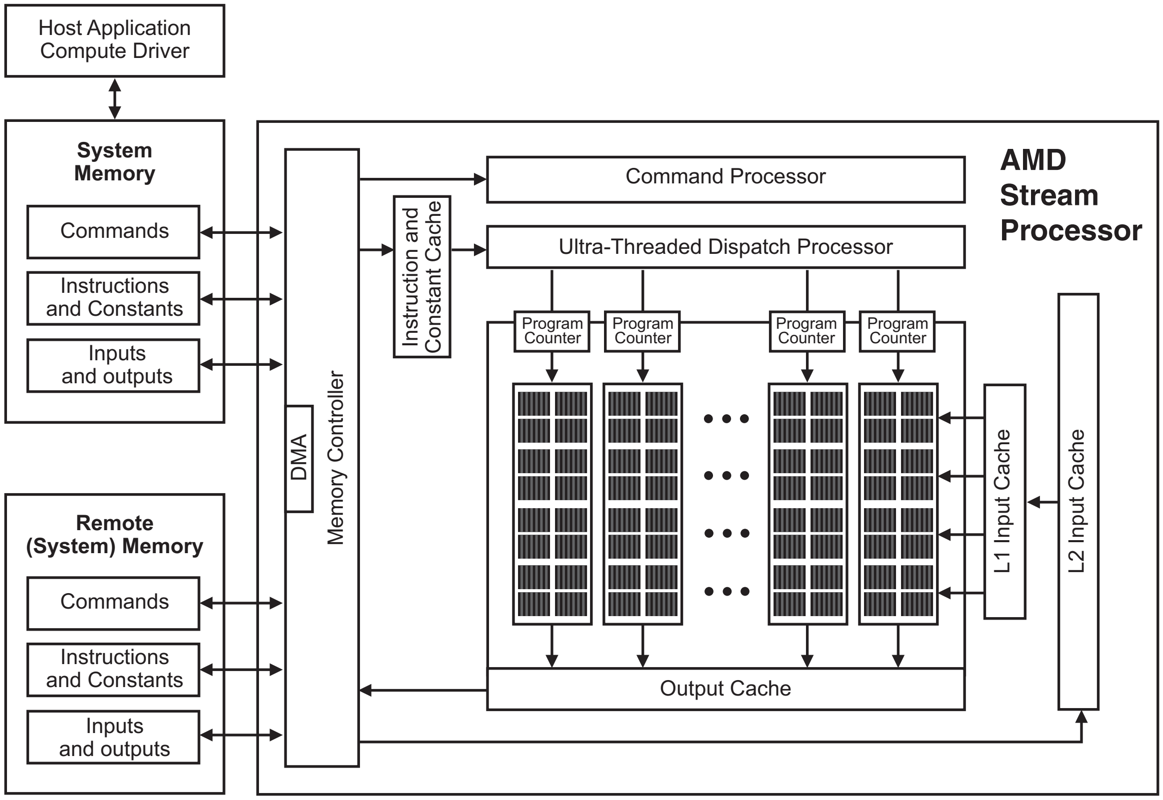

Figure 3.3 AMD Stream Processor Architecture

- The *command processor* reads and initiates commands that the host CPU has sent to the stream processor for execution. The command processor notifies the host when the commands are completed.
- The *stream processor* array is organized as a set of SIMD engines, each independent of the others, that operate in parallel on data streams. The SIMD pipelines can process data or transfer data to and from memory.
- The *memory controller* has direct access to all local memory and host-specified areas of system memory. To satisfy read/write requests, the memory controller performs the functions of a direct-memory access (DMA) controller.
- The stream processor has various caches for data and instructions between the memory controller and the stream processor array.

Kernels are controlled by host commands sent to the stream processors' command processor. These commands typically:

- specify the data domain on which the stream processor operates,
- invalidate and flush caches on the stream processor,
- set up internal base-addresses and other configuration registers,
- request the stream processor to begin execution of a kernel.

The command processor requests a SIMD engine to execute a kernel by passing it an identifier pair (x, y) and the location in memory of the kernel code. The SIMD pipeline then loads instructions and data from memory, begins execution, and continues until the end of the kernel.

Conceptually, each SIMD pipeline maintains a separate interface to memory, consisting of index pairs and a field identifying the type of request (kernel instruction, floating-point constant, integer constant, input read, or output write)[1]. The index pairs for inputs, outputs, and constants are specified by the requesting stream processor instructions from the hardware-maintained kernel state in the pipelines.

The stream processor memory is high-speed DRAM connected to the SIMD engines using a high-speed proprietary interconnect. A host application (running on the CPU) cannot write to stream processor local memory directly, but it can command the stream processor to copy data from system (CPU) memory to stream processor memory, or vice versa.

3.1.2 CAL Programming Model

CAL provides access to the AMD stream processor by offering the runtime and code generation services detailed in the following subsections.

3.1.2.1 Run Time Services

The CAL runtime library, `amdcalrt`, can load and execute the binary image generated by the compiler. The runtime implements:

- *Device Management*: CAL runtime identifies all valid CAL devices on the system. It lets the application query individual device parameters and establish a connection to the device for further operations.
- *Resource Management*: CAL runtime handles the management of all resources, including memory pools available on the system. Memory can be allocated on device local and remote memory subsystems. Data buffers can be efficiently moved between subsystems using DMA transfers.
- *Kernel Loading and Execution*: CAL runtime manages the device state and lets applications set various parameters required for the kernel execution. It provides mechanisms for loading binary images on devices as modules, executing these modules, and synchronizing the execution with the application process.

3.1.2.2 Code Generation Services

The CAL compiler, which is distributed as a separate library (`amdcalcl`) with the CAL SDK, is responsible for the stream processor-specific code generation. The CAL compiler accepts a stream kernel written in one of the supported interfaces and generates the object code for the specified device architecture. The resulting

1. Boolean and double constants are not supported.

CAL object and binary image can be loaded directly on a CAL device for execution (see Figure 3.4).

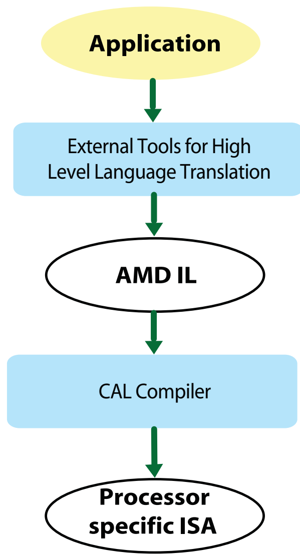

Figure 3.4 CAL Code Generation

The CAL API allows developing stream kernels directly using:

- Device-specific Instruction Set Architecture.
- Pseudo-Assembly languages like AMD's Intermediate Language (IL).

The kernel can be developed in a device-independent manner using the AMD IL. It also is possible to program in a C-like high-level language, such as Brook+. See Appendix B, "The AMD Compute Abstraction Layer (CAL) API Specification" for more information on such tools.

3.1.3 CAL Software Distribution

The distribution software bundle consists of the CAL SDK, which includes platform-specific binaries, header files, sample code, and documentation. This document assumes that the reader has installed the CAL SDK.

On Windows, CAL files are installed in the `%SystemDrive%\Program Files\AMD\AMD CAL x.x.x` directory, where xxx refers to the software version currently installed. The following sections refer to the installation location of the CAL SDK as `$(CALROOT)` and use UNIX-style filepaths for relative paths to specific components.

The SDK contains the following components –

Component	Installation Location
Header files	`$(CALROOT)/include`
Libraries and DLLs (Windows only)	`$(CALROOT)/lib`
Documentation	`$(CALROOT)/doc`
Sample applications	`$(CALROOT)/samples`
Binaries for sample applications	`$(CALROOT)/bin`
Development Tools and Utilities	`$(CALROOT)/tools`, `$(CALROOT)/utilities`

The samples included in the SDK contain simple example programs that illustrate specific CAL features, as well as tutorial programs under `$(CALROOT)/samples/tutorial`. The reader should build and run some of the sample programs to ensure that the system is configured properly and software is installed correctly for CAL development. See the release notes for detailed instructions on the software installation and system configuration.

3.2 CAL Application Programming Interface

The CAL API contains a few C function calls and simple data types used for data specification and processing on the device. The complete list of all functions, along with their C declarations, are in Appendix B, "The AMD Compute Abstraction Layer (CAL) API Specification". Note the following conventions regarding the CAL API:

- All CAL runtime functions use the prefix `cal`. All CAL compiler functions use the prefix `calcl`.
- All CAL utilities use the prefix `calut`.
- All CAL extensions use the prefix `calext`.
- All CAL data types are prefixed with `CAL`. The data types are either `typedefs` to built-in `C` types, or `enums`.
- CAL functions return a status code, `CALresult`. This can be used to check for any internal or usage error within the function. (The exception is disassemble functions, which use `calcldisassemble[image|object]`.) On success, all functions return `CAL_RESULT_OK`. The `calGetErrorString` function provides more information about the error in a human readable string.
- CAL uses opaque handles for internal data structures like `CALdevice` and `CALresource`.

The following sections provide more information about the two main components of the API: the CAL runtime, and the CAL compiler. The complete list of CAL compiler and runtime function calls is in Appendix B, "The AMD Compute Abstraction Layer (CAL) API Specification".

3.2.1 CAL Runtime

The CAL runtime comprises:

- System initialization and query
- Device management
- Context management
- Memory management,
- Program loading
- Program execution

This section covers the first four bulleted items. The last two components, program loading and program execution, are covered in Section 3.2.3, "Kernel Execution," page 3-16.

3.2.1.1 System Initialization and Query

The CAL runtime provides mechanisms for initializing, and shutting down, a CAL system. It also contains methods to query the version of the CAL runtime.

The first CAL routine to be invoked from an application is `calInit`. It initializes the CAL API and identifies all valid CAL devices on the system. Invoking any other CAL function prior to `calInit` results in an error code, `CAL_RESULT_ERROR`. If `calInit` has already been invoked, the routine returns `CAL_RESULT_ALREADY`. Similarly, `calShutdown` must be called before the application exits for the application to shutdown properly. Invoking another CAL routine after `calShutdown` results in a `CAL_RESULT_NOT_INITIALIZED` error.

Query the CAL version on the system with the `calGetVersion` routine. It provides the major and minor version numbers of the CAL release, as well as the implementation instance of the supplied version number.

3.2.1.2 Device Management

The CAL runtime supports managing multiple devices in the system. The CAL API identifies each device in the system with a unique numeric identifier in the range `[0..N-1]`, where `N` is the number of CAL-supported devices on the system. To find the number of stream processors in the system use the `calDeviceGetCount` routine (see the `FindNumDevices` tutorial program). For further information on each device, use the `calDeviceGetInfo` routine. It returns information on the specific device, including the device type and maximum valid dimensions of 1D and 2D buffer resources that can be allocated on this device.

Before any operations can be done on a given CAL device, the application must open a dedicated connection to the device using the `calDeviceOpen` routine. Similarly, the device must be closed before the application exits using the `calDeviceClose` routine (see the `OpenCloseDevice` tutorial program).

The calDeviceOpen routine accepts the numeric identifier for the stream processor that must be opened; when it is open, the routine returns a pointer to the device.

The following code uses these routines.

```
// Initialize CAL system for computation
if(calInit() != CAL_RESULT_OK) ERROR_OCCURRED();

// Query and print the runtime version that is loaded
CALuint version[3];
calGetVersion(&version[0], &version[1], &version[2]);
fprintf(stderr, “CAL Runtime version %d.%d.%d\n”,
                 version[0], version[1], version[2]);

// Query the number of devices on the system
CALuint numDevices = 0;
if(calDeviceGetCount(&numDevices) != CAL_RESULT_OK) ERROR_OCCURRED();

// Get the information on the 0th device
CALdeviceinfo info;
if(calDeviceGetInfo(&info, 0) != CAL_RESULT_OK) ERROR_OCCURRED();

switch(info.target)
{
   case CAL_TARGET_600:
      fprintf(stdout, "Device Type = GPU R600\n");
      break;
   case CAL_TARGET_670:
      fprintf(stdout, "Device Type = GPU RV670\n");
      break;
}

// Opening the 0th device
CALdevice device = 0;
if(calDeviceOpen(&device, 0) != CAL_RESULT_OK) ERROR_OCCURRED();

// Use the device
// ……………

// Closing the device
calDeviceClose(device);

// Shutting down CAL
if(calShutdown() != CAL_RESULT_OK) ERROR_OCCURRED();
```

The calDeviceGetInfo routine provides basic information. For more detailed information about the device, use the calDeviceGetAttribs routine. It returns a C struct of type CALdeviceattribs with fields of information on the stream processor ASIC type, available local and remote RAM sizes, and stream processor clock speed. Note, however, that setting struct.struct_size to the size of CALdeviceattribs must be done before calling calDeviceGetAttribs.

3.2.1.3 Context Management

To execute a kernel on a CAL device, the application must have a valid CAL context on that device (see the `CreateContext` tutorial program). A CAL context is an abstraction representing all the device states that affect the execution of a CAL kernel. A CAL device can have multiple contexts, but the same context cannot be shared by more than one CAL device. For multi-threaded applications, each CPU thread must use a separate CAL context for communicating with the CAL device (see Figure 3.5; also, see Section 3.7, "Advanced Topics," for more information).

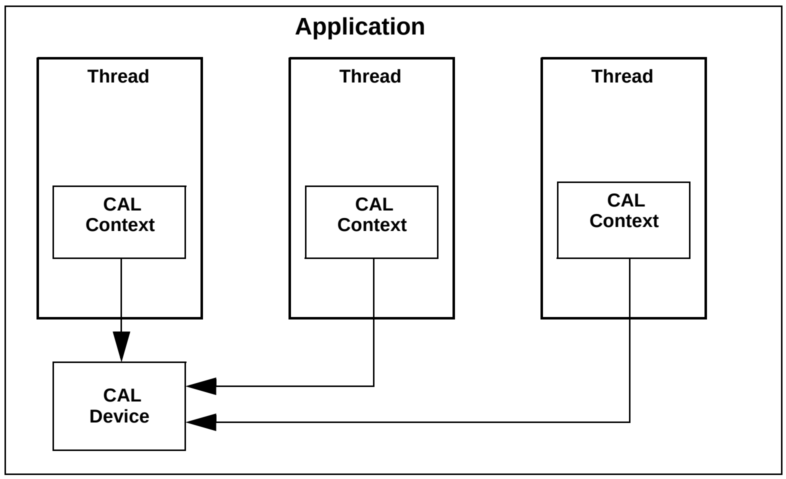

Figure 3.5 Context Management for Multi-Threaded Applications

A CAL context can be created on the specified device using the `calCtxCreate` routine. Similarly, a context can be deleted using the `calCtxDestroy` routine.

```
// Create context on the device
CALContext ctx;
if(calCtxCreate(&ctx, device) != CAL_RESULT_OK) ERROR_OCCURRED();
// Destroy the context
if(calCtxDestroy(ctx) != CAL_RESULT_OK) ERROR_OCCURRED();
```

3.2.1.4 Memory Management

All CAL devices have access to local and remote memory subsystems through CAL kernels running on the device. These discrete memory subsystems are known collectively as memory pools. In the case of stream processors, local memory corresponds to the high-speed video memory located on the graphics board. Remote memory corresponds to memory that is not local to the given device but still visible to a set of devices (see Figure 3.6). To find the total size of each memory pool available to a given device, use the `calDeviceGetAttribs` routine.

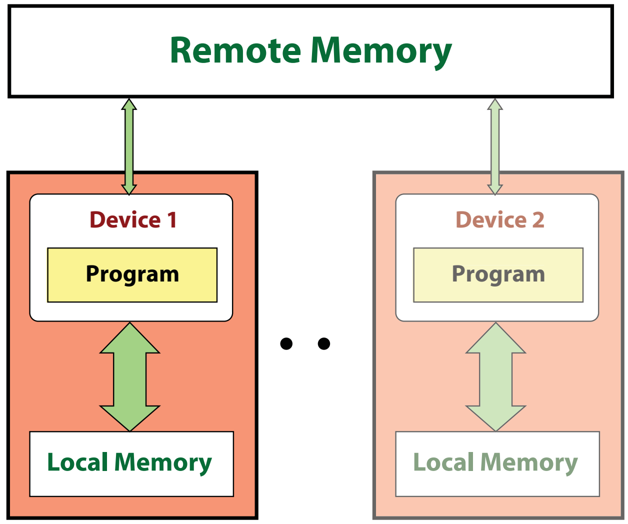

Figure 3.6 Local and Remote Memory

The most common case of remote memory that is accessible from the stream processors is the system memory. In this case, the stream kernel accesses memory over the PCIe bus. This access usually is slower and incurs a higher latency compared to local memory. Performance is dependent on the characteristics and architectural topology of the host RAM, processor, and the PCIe controller on the system.

The following steps allocate, initialize and use memory buffers in a CAL kernel:

- Allocate memory resources with desired parameters and memory subsystem.
- Map input and constant resources to application address space, and initialize contents on the host.
- Provide each resource with context-specific memory handles.
- Bind memory handles to corresponding parameter names in the kernel.

3.2.1.5 Resources

In CAL, all physical memory blocks allocated by the application for use in stream kernels are referred to as resources. These blocks can be allocated as one-dimensional or as two-dimensional arrays of data. The data type and format for each element in the array must be specified at the time of resource allocation (see the `CreateResource` tutorial program).

The supported formats include:

- 8-, 16-, and 32-bit, signed and unsigned integer types with 1, 2, and 4 components per element, as well as
- 32- and 64-bit floating point types with 1, 2, and 4 components per element.

The formats are specified using the `CALformat` enumerated type. The enums use the naming syntax CAL_FORMAT_*type*_*n*, where *type* is the data type and *n* is the number of components per element. For example, CAL_FORMAT_UBYTE_4 represents an element with four 4 8-bit unsigned integer values per element.

Note: 4-component 64-bit floating point types are not supported with this version of the CAL release.

Memory can be allocated locally (stream processor memory) or remotely (system memory). In the case of remote allocation, the CAL API lets the application control the list of devices that can access the resource directly. Remote memory can serve as a mechanism for sharing memory resources between multiple devices. This prevents the application from having to create multiple copies of the data.

Local resources can be allocated using the calResAllocLocalnD routines, where n is the dimension of the array. Currently, n can be only 1 or 2. The routine requires the application to pass the CALDevice on which the resource is allocated along with other parameters such as width, height, format, etc. Similarly, remote resources are allocated using the calResAllocRemotenD routines and require the list of CAL devices that can share the remote resource. The allocated resource is visible only to these devices. On successful completion of the allocation, the CAL API returns a pointer to the newly allocated CALResource. To deallocate a resource, use the calResFree routine.

The following code allocates a 2D resource of 32-bit floating point values on the specified CAL device.

```
// Allocate 2D array of FLOAT_1 data
CALresource resLocal = 0;
if(calResAllocLocal2D(&resLocal, device, width, height,
                       CAL_FORMAT_FLOAT_1, 0) != CAL_RESULT_OK)
ERROR_OCCURRED();

// Do the computations
// ……………

// De-allocate the resource
if(calResFree(resLocal) != CAL_RESULT_OK) ERROR_OCCURRED();
```

CAL memory is used as inputs, outputs, or constants to CAL kernels. For inputs and constants, first initialize the contents of the memory buffer from the host application. One way to do this is to map the memory to the application's address space using the calResMap routine. The routine returns a host-side memory pointer that the application can dereference; the application then initializes the buffer. The routine also returns the pitch of the data buffer, which must be considered when dereferencing this data. The pitch corresponds to the number of elements in each row of the resource. This usually is equal to, or greater than, the width specified in the allocation routine. The size of the memory buffer allocated is given by:

```
Allocated Buffer Size = Pitch * Height * Number of components *
                        Size of data type
```

The following code demonstrates how to use calResMap to initialize the resource allocated above.

```
// Map the memory handle to CPU pointer
float *dataPtr = NULL;
CALuint pitch = 0;
if(calResMap((CALVoid **)&dataPtr, &pitch, resLocal, 0) !=
                CAL_RESULT_OK) ERROR_OCCURRED();

// Initialize the data values
for(int i = 0; i < height; i++)
{
    // Note the use of the pitch returned by calResMap to properly
    // offset into the memory pointer
    float* tmp = &dataPtr[i * pitch];

    for (int j = 0; j < width; j++)
    {
        // At this place depending on the format (1,2,4) we can
        // specify relevant values i.e.
        // For FLOAT_1, we should initialize temp[j]
        // For FLOAt_2, we should initialize temp[2*j] & temp[2*j + 1]
        // For FLOAT_4, we should initialize temp[4*j], temp[4*j + 1],
        // temp[4*j + 2] & temp[4*j + 3]
            tmp[j] = (float)(i * width + j);
    }
}

// Unmap the memory handle
if(calResUnmap(resLocal) != CAL_RESULT_OK) ERROR_OCCURRED();
```

Note that a mapped resource cannot be used in a CAL kernel; the resource must be unmapped using calResUnmap before being used as shown above.

3.2.1.6 Memory Handles

Once a resource has been allocated, it must be bound to a given CAL context before being used in a CAL kernel. CAL resources are not context-specific. Hence, they first must be mapped to the given context's address space before being addressed by that context. This is done using the calCtxGetMem routine. When this is done, the routine returns a context-specific memory handle to the resource. This handle can be used for subsequent operations, such as reading from, and writing to, the resource. Once the memory handle is no longer needed, the handle can be released using the calCtxReleaseMem routine.

```
// Map the given resource to a new memory handle for this context
CALmem memLocal = 0;
if(calCtxGetMem(&memLocal, ctx, resLocal) != CAL_RESULT_OK)
    ERROR_OCCURRED();

// Use the memory handle
// ………………

// Release the resource to context mapping
if(calCtxReleaseMem(ctx, memLocal) != CAL_RESULT_OK)
ERROR_OCCURRED();
```

The SetupData routine in the basic tutorial program implements the steps required to allocate, initialize, and use memory buffers in a kernel. The last step of binding memory handles to kernel names and parameter names is explained in Section 3.2.3, "Kernel Execution."

3.2.2 CAL Compiler

The CAL compiler provides a high-level runtime interface for compiling stream kernels written in one of the supported programming interfaces. The compiler can be invoked either at runtime or offline. Invoking them at runtime typically happens during kernel development when the developer constantly modifies the kernel and tests the output results. Invoking the offline compiler is suitable for production-class applications, including kernels that have already been developed and are loaded and invoked only at runtime. This mechanism prevents the overhead of compiling the kernel each time the application is executed.

AMD provides other useful tools that can be used for fast and easy development of efficient stream kernels. See Appendix B, "The AMD Compute Abstraction Layer (CAL) API Specification" and Section 1.1.4, "GPU ShaderAnalyzer," page 1-10, for more information.

3.2.2.1 Compilation and Linking

The CAL compiler accepts the kernel in one of the supported programming interfaces and generates a binary object specific to a given target CAL device using calclCompile (see the CompileProgram tutorial program). The routine requires the application to specify, as arguments, the interface type and the target device architecture for the resulting binary object, along with the C-style string for the stream kernel. Once compiled, the object must be linked into a binary image using calclLink, which generates this image. The binary object and image are returned as the handles CALobject and CALimage, respectively.

Note the following guidelines for the CAL compiler API:

- Only the AMD IL and the stream processor-specific Instruction Set Architecture (ISA) are supported as the runtime programming interfaces by calclCompile.
- The target device architecture supported includes AMD stream processors listed under the CALtarget enumerated type.

The following code shows the use of the CAL compiler API for querying the compiler version, compiling a minimal AMD IL kernel and linking the resulting object into the final binary image. Note the use of the calclFreeObject and calclFreeImage routines for deallocating the memory allocated by the CAL compiler for the program object and binary image.

```
// Kernel string
const char ilKernel[] =
"il_ps_2_0 \n"
// other instructions
"ret_dyn \n"
"end \n";

// Query and print the compiler version that is loaded
CALuint version[3];
calclGetVersion(&version[0], &version[1], &version[2]);
fprintf(stderr, "CAL Compiler version %d.%d.%d\n",
                 version[0], version[1], version[2]);

// Compile the IL kernel
CALobject object = NULL;
if(calclCompile(&object, CAL_LANGUAGE_IL, ilKernel, CAL_TARGET_670) !=
                 CAL_RESULT_OK))
    ERROR_OCCURRED();

// Link the objects into CAL image
CALimage image = NULL;
if(calclLink (&image, &object, 1) != CAL_RESULT_OK))
    ERROR_OCCURRED();

// Use the CAL runtime API to load and run the kernel
// ……………

// Free the object
calclFreeObject(object);

// Free the image
calclFreeImage(image);
```

3.2.2.2 Stream Processor ISA

The CAL compiler compiles and optimizes the input AMD IL pseudo-assembly to generate the stream processor-specific ISA. The developer can use the AMD IL or the stream processor ISA for developing the kernel. Figure 3.7 illustrates the sequence of steps used during the compilation process. In the latter case, `calclAssembleObject` is used to create the CALObject from the stream processor ISA. Note that this routine performs no optimizations, and the resulting binary is a direct mapping of the specified stream processor ISA. When using the AMD IL, the conversion from AMD IL to the stream processor ISA is done internally by the CAL compiler. This process is transparent to the application. However, reviewing and understanding the stream processor ISA can be extremely useful for program debugging and performance profiling purposes. To get the stream processor ISA for a given `CALimage`, use the `calclDisassembleImage` routine; for CAL objects, use `calclDisassembleObject`.

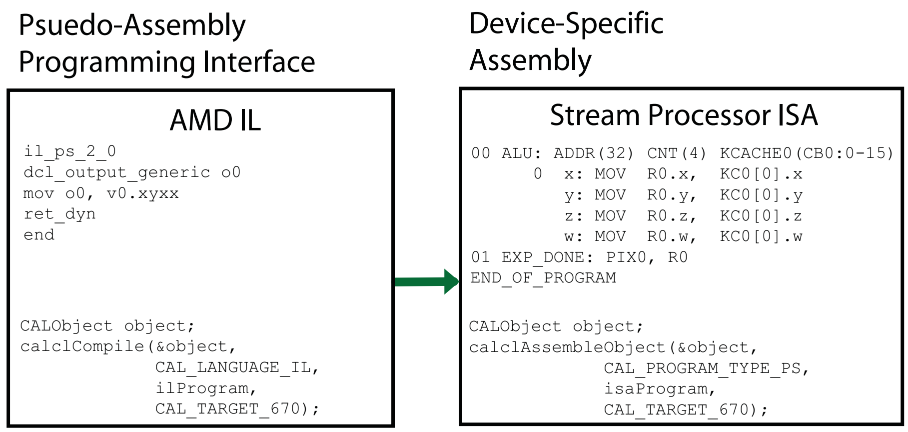

Figure 3.7 Kernel Compilation Sequence

3.2.2.3 High Level Kernel Languages

High-level kernel languages, such as Brook+, provide advantages during kernel development such as ease of development, code readability, maintainability, and reuse. AMD-specific interfaces such as AMD IL provide access to lower-level features in the device, permitting improved features and performance tuning. To facilitate leveraging the advantages of each programming interface, AMD provides offline tools that aid with high-level kernel development while providing low-level control by exposing the AMD IL and the stream processor ISA. For example, developers can use Brook+ to develop their kernels, then generate the equivalent AMD IL using offline tools provided by AMD (see Appendix B, "The AMD Compute Abstraction Layer (CAL) API Specification," and Section 1.1.4, "GPU ShaderAnalyzer," page 1-10). The generated AMD IL kernel then can be passed to the CAL compiler, with any required modifications, for generating the binary image.

3.2.3 Kernel Execution

Once the application has initialized the various components (including the device, memory buffers and program binary), it is ready to execute the kernel on the device. Kernel execution on a CAL device consists of the following high level steps: module loading, parameter binding and kernel invocation (see the `basic` tutorial program).

3.2.3.1 Module Loading

Once a CAL image has been linked, it must be loaded as an executable module by the CAL runtime using the `calModuleLoad` routine. For execution, the runtime must specify the entry point within the module. This can be queried using the function name in the original kernel string. Currently, the function name is always set to `main`. The following code is an example of loading an executable module.

```
// Load CAL image as a runtime module for this context
CALmodule module = 0;
if(calModuleLoad(&module, ctx, image) != CAL_RESULT_OK)
    ERROR_OCCURRED();

// Query the entry point in the module for the function “main”
CALfunc entry = 0;
if(calModuleGetEntry(&entry, ctx, module, "main") != CAL_RESULT_OK)
    ERROR_OCCURRED();
```

3.2.3.2 Parameter Binding

The CAL runtime API also provides an interface to set up various parameters (inputs and outputs) required by the CAL API for proper execution. CAL identifies each parameter in the module by its variable name in the original kernel string. These variables are AMD IL-style names for inputs (i#), outputs (o#), and constant buffers (cb#), as shown in the following code. The runtime provides a routine, `calModuleGetName`, that allows retrieving a handle from each of the variables in the module as `CALname`. Here, `x#[ ]` is for the scratch buffer, `g[ ]` is for the global buffer, i# is for the input buffer, and o# is for the output buffer. These parameter name handles subsequently can be bound to specific memory handles using `calCtxSetMem`, then used by the CAL kernel at runtime. The following code is an example of binding parameters.

```
// Query the variable names for input 0 and output 0
CALname input = 0, output = 0;
if(calModuleGetName(&input, ctx, module, "i0") != CAL_RESULT_OK ||
    calModuleGetName(&output, ctx, module, "o0") != CAL_RESULT_OK)
      ERROR_OCCURRED();

CALmem inputMem = 0, outputMem = 0;

// Bind resources to memory handles for this context
// ……………

// Bind the parameters to memory handles
if(calCtxSetMem(ctx, input, inputMem) != CAL_RESULT_OK ||
    calCtxSetMem(ctx, output, outputMem) != CAL_RESULT_OK)
      ERROR_OCCURRED();
```

3.2.3.3 Kernel Invocation

Kernels are executed over a rectangular region of the output buffer called the *domain of execution*. The kernel is launched using the `calCtxRunProgram` routine, which specifies the context, entry point, and domain of execution. The routine returns an event identifier for this invocation. The `calCtxRunProgram` routine is a non-blocking routine and returns immediately. The application thread calling this routine is free to execute other tasks while the computation is being done on the CAL device. Alternatively, the application thread can use a busy-wait loop to keep polling on the completion of the event by using the `calCtxIsEventDone` routine. The following code is an example of invoking a kernel.

```
// Setup the domain for execution
CALdomain domain = {0, 0, width, height};

// Event ID corresponding to the kernel invocation
CALevent event = 0;

// Launch the CAL kernel on the given domain
if(calCtxRunProgram(&event, ctx, entry, &domain) != CAL_RESULT_OK)
    ERROR_OCCURRED();

// Wait on the event for kernel completion
while(calCtxIsEventDone(ctx, event) == CAL_RESULT_PENDING);
```

When the above loop returns, the stream kernel has finished execution, and the output memory can be dereferenced (using `calResMap`) to access the output results. Note the following:

- The `domain` (domain of execution) is a subset of the output buffer. The stream processor creates a separate thread for each (x,y) location in the domain of execution.
- For improved performance, `calCtxRunProgram` does not immediately dispatch the program for execution on the stream processor. To force the dispatch, the application must call `calCtxIsEventDone` and `calCtxFlush` on the corresponding event.

3.3 HelloCAL Application

This section provides a simple example that combines the concepts covered in previous sections in the form of a `HelloCAL` application. This program demonstrates the following components:

- Initializing CAL
- Compiling and loading a stream kernel
- Opening a connection to a CAL device
- Allocating memory
- Specifying kernel parameters including inputs, outputs, and constants
- Executing the CAL kernel

`HelloCAL` uses a CAL kernel written in AMD IL; this shows the actions taken when running a CAL application. The kernel reads from one input, multiplies the resulting value by a constant, and writes to one output. In vector notation, the computation can be represented as:

```
Out(1:N) = In(1:N) * constant;
```

3.3.1 Code Walkthrough

This section analyzes the major blocks of code in HelloCAL. The code provided in this section is a complete application. The reader can copy the code examples into a separate C++ file to compile and run it.

3.3.1.1 Basic Infrastructural Code

The following code contains the basic infrastructural code, including headers used by the application. Note that cal.h and calcl.h are shipped as part of the standard CAL headers. Building HelloCAL requires the amdcalrt and amdcalcl libraries.

```
////////////////////////////////////////////////////////////////////
//! Header files
////////////////////////////////////////////////////////////////////
#include "cal.h"
#include "calcl.h"
#include <string>
```

The reader must have a basic understanding of AMD IL. The *AMD IL Programmer's Manual* provides a detailed specification on the AMD IL interface.

3.3.1.2 Defining the Stream Kernel

The following code defines the stream kernel written in AMD IL.

This stream kernel:

- Looks up the 0'th input buffer via the 0'th sampler, using sample_resource(*n*)_sampler(*m*) instruction. The current fragment's position, v0.xy, is the index into the input buffer. It stores the resulting value in temporary register r0.
- Multiplies the value in r0 with the constant cb0[0], and writes the resulting value to output buffer o0.

```
//////////////////////////////////////////////////////////////////////////
//! Device Kernel to be executed on the GPU
//////////////////////////////////////////////////////////////////////////
//! IL Kernel
std::string kernelIL =
"il_ps_2_0\n"
"dcl_input_position_interp(linear_noperspective) vWinCoord0.xy__\n"
"dcl_output_generic o0\n"
"dcl_cb cb0[1]\n"
"dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fm
tw(float)\n"
"sample_resource(0)_sampler(0) r0, vWinCoord0.xyxx\n"
"mul o0, r0, cb0[0]\n"
"end\n";

    };
```

3.3.1.3 Application Code

The following code contains the actual application code that initializes CAL, queries the number of devices on the given system, and opens a connection to the 0'th CAL device. The application then creates a CAL context on this device.

```
/////////////////////////////////////////////////////////////////////
//! Main function
/////////////////////////////////////////////////////////////////////
int main(int argc, char** argv)
{
    // Initializing CAL
    calInit();
    //-------------------------------------------------------------
    // Querying and opening device
    //-------------------------------------------------------------
    // Finding number of devices
    CALuint numDevices = 0;
    calDeviceGetCount(&numDevices);

    // Opening device
    CALdevice device = 0;
    calDeviceOpen(&device, 0);

    // Querying device info
    CALdeviceinfo info;
    calDeviceGetInfo(&info, 0);

    // Creating context w.r.t. to opened device
    CALcontext ctx = 0;
calCtxCreate(&ctx, device);
```

3.3.1.4 Compile the Stream Kernel and Link Generated Object

The following code compiles the stream kernel using the `calcl` compiler; it then links the generated object files into a `CALimage`. Note that the stream kernel is being compiled for the AMD device queried to be present on the system using the `calDeviceGetInfo` routine. Also note that the `calclLink` routine can be used to link multiple object files into a single binary image.

```
//-------------------------------------------------------------------
// Compiling Device Kernel
//-------------------------------------------------------------------
CALobject obj = NULL;
CALimage image = NULL;
CALlanguage lang = CAL_LANGUAGE_IL;
std::string kernel = kernelIL;
std::string kernelType = "IL";

if (calclCompile(&obj, lang, kernel.c_str(), info.target) !=
     CAL_RESULT_OK)
{
    fprintf(stdout, "Kernel compilation failed. Exiting.\n");
    return 1;
}

if (calclLink(&image, &obj, 1) != CAL_RESULT_OK)
{
    fprintf(stdout, "Kernel linking failed. Exiting.\n");
    return 1;
}
```

3.3.1.5 Allocate Memory

The following code allocates memory for various buffers to be used by the CAL API. Note that:

- All memory buffers in the application are allocated locally to the opened CAL device. In the case of stream processors, this memory corresponds to stream processor memory.
- The *input and output buffers* contain one-element float values. CAL also allows elements with one, two, and four data values per element arranged in an interleaved manner. For example, CAL_FORMAT_FLOAT4 stores four floating point values per element in the buffer. This can be extremely useful in certain algorithms since it allows reading multiple values using a single read instruction.
- The resources must be mapped to CPU memory handles before they can be referenced in the application. The pitch of the buffer must be considered while dereferencing the data pointer.
- Any constants required by the kernel can be passed as a one-dimensional array of data values. This array must be allocated, mapped, and initialized similar to the way input buffers are handled. In the following code, the *constant buffer* is allocated in remote memory.

```
//------------------------------------------------------------------------
// Allocating and initializing resources
//------------------------------------------------------------------------
// Input and output resources
CALresource inputRes = 0;
CALresource outputRes = 0;

calResAllocLocal2D(&inputRes, device, 256, 256, CAL_FORMAT_FLOAT_1, 0);
calResAllocLocal2D(&outputRes, device, 256, 256, CAL_FORMAT_FLOAT_1, 0);

// Constant resource
CALresource constRes = 0;
calResAllocRemote1D(&constRes, &device, 1, 1, CAL_FORMAT_FLOAT_4, 0);

// Setup input buffer – map resource to CPU, initialize values, unmap resource
float* fdata = NULL;
CALuint pitch = 0;
CALmem inputMem = 0;

// Mapping resource to CPU
calResMap((CALvoid**)&fdata, &pitch, inputRes, 0);
for (int i = 0; i < 256; ++i)
{
    float* tmp = &fdata[i * pitch];
    for (int j = 0; j < 256; ++j)
    {
            tmp[j] = (float)(i * pitch + j);
    }
}
calResUnmap(inputRes);

// Setup const buffer – map resource to CPU, initialize values, unmap resource
float* constPtr = NULL;
CALuint constPitch = 0;
CALmem constMem = 0;
```

```
calResMap((CALvoid**)&constPtr, &constPitch, constRes, 0);
constPtr[0] = 0.5f, constPtr[1] = 0.0f;
constPtr[2] = 0.0f; constPtr[3] = 0.0f;
calResUnmap(constRes);

// Mapping output resource to CPU and initializing values
void* data = NULL;

// Getting memory handle from resources
CALmem outputMem = 0;
calResMap(&data, &pitch, outputRes, 0);
memset(data, 0, pitch * 256 * sizeof(float));
calResUnmap(outputRes);

// Get memory handles for various resources
calCtxGetMem(&constMem, ctx, constRes);
calCtxGetMem(&outputMem, ctx, outputRes);
calCtxGetMem(&inputMem, ctx, inputRes);
```

3.3.1.6 Preparing the Stream Kernel for Execution

The following code prepares the stream kernel for execution. The CAL image is first loaded into a CALmodule. Subsequently, the names for various parameters used in the stream kernel, including the input, output, and constant buffers, are queried from the module. The names are then bound to appropriate memory handles corresponding to these parameters. Finally, the kernel's domain of execution is set up. In this case, the domain is the same as the dimensions of the output buffer. This is the most commonly used scenario, even though CAL allows specifying domains that are subsets of the output buffers. Note that all the settings mentioned above are collectively called the kernel state and are associated with the current CAL context.

```
//-------------------------------------------------------------------
// Loading module and setting domain
//-------------------------------------------------------------------

// Creating module using compiled image
CALmodule module = 0;
calModuleLoad(&module, ctx, image);

// Defining symbols in module
CALfunc func = 0;
CALname inName = 0, outName = 0, constName = 0;

// Defining entry point for the module
calModuleGetEntry(&func, ctx, module, "main");
calModuleGetName(&inName, ctx, module, "i0");
calModuleGetName(&outName, ctx, module, "o0");
calModuleGetName(&constName, ctx, module, "cb0");

// Setting input and output buffers
// used in the kernel
calCtxSetMem(ctx, inName, inputMem);
calCtxSetMem(ctx, outName, outputMem);
calCtxSetMem(ctx, constName, constMem);

// Setting domain
CALdomain domain = {0, 0, 256, 256};
```

3.3.1.7 Kernel Execution

Once the above state has been set, the stream kernel can be launched using the `calCtxRunProgram` routine. The function `main` in the stream kernel is queried from the module and specified as the entry point during kernel launch. The `calCtxRunProgram` function returns an event identifier, `CALevent`, for the current kernel launch. This identifier can determine if the event has completed. Note that if a certain state setting required by the kernel is not set up before launching the kernel, the `calCtxRunProgram` call fails.

```
//---------------------------------------------------------------
// Executing kernel and waiting for kernel to terminate
//---------------------------------------------------------------

// Event to check completion of the kernel
CALevent e = 0;
calCtxRunProgram(&e, ctx, func, &domain);

// Checking whether the execution of the kernel is complete or not
while (calCtxIsEventDone(ctx, e) == CAL_RESULT_PENDING);

// Reading output from output resources
calResMap((CALvoid**)&fdata, &pitch, outputRes, 0);
for (int i = 0; i < 8; ++i)
{
   float* tmp = &fdata[i * pitch];
   for(int j = 0; j < 8; ++j)
   {
      printf("%f ", tmp[j]);
   }
   printf("\n");
}
calResUnmap(outputRes);
```

When the `calCtxIsEventDone` loop ends, the stream kernel has finished execution. The output memory can be dereferenced (using `calMemResMap`) to access the results in system memory.

3.3.1.8 De-Allocation and Releasing Connections

After the kernel execution, de-allocate the various resources, and release the connections to the device and corresponding contexts to exit the application cleanly. The following code demonstrates this process. Resource de-allocation includes:

- unbinding of memory handles (setting handle identifier as 0 in `calCtxSetMem`),
- releasing memory handles (`calCtxReleaseMem`), and
- de-allocating resources (`calResFree`).

Devices and contexts can be released by destroying the context (`calCtxDestroy`) and closing the device (`calDeviceClose`).

```
//-------------------------------------------------------------------
// Cleaning up
//-------------------------------------------------------------------

// Unloading the module
calModuleUnload(ctx, module);

// Freeing compiled kernel binary
calclFreeImage(image);
calclFreeObject(obj);

// Releasing resource from context
calCtxReleaseMem(ctx, inputMem);
calCtxReleaseMem(ctx, constMem);
calCtxReleaseMem(ctx, outputMem);

// Deallocating resources
calResFree(outputRes);
calResFree(constRes);
calResFree(inputRes);

// Destroying context
calCtxDestroy(ctx);
// Closing device
calDeviceClose(device);

// Shutting down CAL
calShutdown();

return 0;
}
```

Remember that `calShutdown` must be the last CAL routine to be called by the application.

3.4 Performance Optimizations

A main objective of CAL is to facilitate high-performance computing by leveraging the power of AMD Stream Processors. It is important to understand the performance characteristics of these devices to achieve the expected performance. The following subsections provide information for developers to fine-tune the performance of their CAL applications.

3.4.1 Arithmetic Computations

Modern computational devices are extremely fast at arithmetic computations due to the large number of stream cores. This is true for floating point and integer arithmetic operations. For example, the peak floating point computation capability of a device is given by:

```
Peak GPU FLOPs = Number of FP stream cores * FLOPS per stream core unit
```

The AMD RV670 stream processor has 320 stream cores. Each of these is capable of executing one MAD (multiply and add) instruction per clock cycle. If the clock rate on the stream cores is 800 MHz, the FLOPs per stream core are given by:

```
FLOPS per Stream Core = Clock rate * Number of FP Ops per clock
              = 800 * 10^6 * 2
              = 1.6 GigaFLOPS
```

Thus, the cumulative FLOPS of the stream processor is given by:

```
Peak GPU FLOPS = 320 * 1.6 = 512 GigaFLOPS
```

The stream processor is extremely powerful at stream core computations. The CAL compiler optimizes the input AMD IL so the stream cores are used efficiently. The compiler also removes unnecessary computations in the kernel and optimizes the use of processor resources like temporary registers. Note that no optimizations are done if the kernel is written in the device ISA.

3.4.2 Memory Considerations

Stream kernels access memory for reading from inputs and writing to outputs. Getting the maximum performance from a CAL kernel usually means optimizing the memory access characteristics of the kernel. The following subsections discuss these considerations.

3.4.2.1 Local and Remote Resources

Accessing local memory from the device is typically more efficient due to the low-latency, high-bandwidth interconnect between the device and local memory. To minimize the effect on performance, memory intensive kernels can:

- Copy the input data buffers to local memory.
- Execute the stream kernel by reading from local inputs and writing to local outputs.
- Copy the outputs to application's address space in system memory.

3.4.2.2 Cached Remote Resources

A typical CAL application initializes input data in system memory. In some cases, the data must be processed by the CPU before being sent to the stream processor for further processing. This processing requires the CPU to read from, and write to, system memory. Here, it might be more efficient to request CAL to allocate this remote (CPU) memory from cached system memory for faster processing of data from the CPU. This can be done by specifying the `CAL_RESALLOC_CACHEABLE` flag to `calResAllocRemote*` routines, as shown in the following code.

```
// Allocate cached 2D remote resource
CALresource cachedRes = 0;
if(calResAllocRemote2D(&cachedRes, &device, 1, width, height,
        CAL_FORMAT_FLOAT_4, CAL_RESALLOC_CACHEABLE != CAL_RESULT_OK)
{
    fprintf(stdout, "Cached resources not available on device
%u\n",
        device);
    return -1;
}
```

When using cached system memory, note that:

- By default, the memory allocated by CAL is uncached system memory if the flag passed to `calResAllocRemote*` is zero.
- Uncached memory typically gives better performance for memory operations that do not use the CPU; for example, DMA (direct memory access) operations used to transfer data from system memory to stream processor local memory, and vice-versa. Note that accessing uncached memory from the CPU degrades performance.
- The application must verify the value returned by `calResAllocRemote*` to see if the allocation succeeded before using the CAL resource. When requesting cached system memory, `calResAllocRemote*` fails and returns a NULL resource handle when:
  - The host system on which the application is running does not support cached system memory.
  - The amount of cached system memory requested is not available. The maximum size of cached memory available to an application typically is limited by the underlying operating system. The exact value can be queried using the `calDeviceGetAttribs` routine. The value is stored as `cachedRemoteRAM` under `CALdeviceattribs`.

3.4.2.3 Direct Memory Access (DMA)

Direct memory access (DMA) allows devices attached to the host sub-system to access system memory directly, independent of the CPU (see the `DownloadReadback` tutorial program). Depending on the available system interconnect between the system memory and the stream processor, using DMA can help improved data transfer rates when moving data between the system memory and stream processor local memory. As seen in Figure 3.3, the AMD stream processor contains a dedicated DMA unit for these operations. This DMA unit can run asynchronously from the rest of the stream processor, allowing parallel data transfers when the SIMD engine is busy running a previous stream kernel.

Applications can request a DMA transfer from CAL using the `calMemCopy` routine when copying data buffers between remote (system) and local (stream processor) memory, as shown in the following code.

```
int
copyData(CALcontext ctx, CALmem input, CALmem output)
{
   // Initiate the DMA transfer – input is a remote resource
   // and output is a device local resource
   CALevent e;
   CALresult r = calMemCopy(&e, ctx, input, output, 0);
   if (r != CAL_RESULT_OK)
   {
      fprintf(stdout, "Error occurred in calMemCopy\n");
      return -1;
}

   // Potentially do other stuff except for dereferencing input or
   // output resources
   // ……………

   // If the routine did not return any error, wait for the DMA
   // to finish
   if (r == CAL_RESULT_OK)
{
   while (calCtxIsEventDone(ctx, e) == CAL_RESULT_PENDING);
}
```

3.4.3 Asynchronous Operations

The calCtxRunProgram and calMemCopy routines are non-blocking and return immediately. Both return a CALevent that can be polled using calCtxIsEventDone to check for routine completion. Since these routines are executed on dedicated hardware units on the stream processor, namely the DMA unit and the Stream Processor array, the application thread is free to perform other operations on the CPU in parallel.

For example, consider an application that must perform CPU computations in the application thread and also run another kernel on the stream processor. The following code shows one way of doing this.

```
// Launch GPU kernel
CALevent e;
if(calCtxRunProgram(&e, ctx, func, &rect) != CAL_RESULT_OK)
   fprintf(stderr, "Error in run kernel\n");

// Wait for the GPU kernel to finish
while(calCtxIsEventDone(ctx, e) == CAL_RESULT_PENDING);

// Perform CPU operations _after_ the GPU kernel is complete
performCPUOperations();

// Map the output resource to application data pointer
calResMap((CALvoid**)&fdata, &pitch, outputRes, 0);
```

The following code implements the same operations as above, but probably finishes more quickly since it executes the CPU operations in parallel with the stream kernel.

```
// Launch GPU kernel
CALevent e;
if(calCtxRunProgram(&e, ctx, func, &rect) != CAL_RESULT_OK)
   fprintf(stderr, "Error in run kernel\n");

// Force a dispatch of the kernel to the device
calCtxIsEventDone(ctx, e);

// Perform CPU operations _in parallel_ with the GPU kernel
execution
performCPUOperations();

// Wait for the GPU kernel to finish
while(calCtxIsEventDone(ctx, e) == CAL_RESULT_PENDING);

// Map the output resource to application data pointer
calResMap((CALvoid**)&fdata, &pitch, outputRes, 0);
```

Note that the above code assumes that the CPU operations in `performCPUOperations()` do not use, or depend upon, any of the output values computed in the stream kernel. If `calResMap` is called before the `calCtxIsEventDone` loop, the above code might generate incorrect results. The same logic mentioned above can be applied for all combinations of DMA transfers, stream kernel execution, and CPU computations.

When using the CAL API, the application must correctly synchronize operations between the stream processor, DMA engine, and CPU. The above example shows how developers can use the CAL API to improve application performance with a proper understanding of the data dependencies in the application and the underlying system's architecture.

These DMA transfers can be asynchronous. The DMA engine executes each transfer separately from the command queue. DMA calls are executed immediately; and the order of DMA calls and command queue flushes is guaranteed.

DMA transfer execute concurrently with other system or stream processor operations; however, data is not guaranteed to be ready until the DMA engine signals that the event or transfer is completed. The application can query the hardware for DMA event completion. DMA transfers can be another source of parallelization.

3.5 Tutorial Application

This section uses a very common problem in Linear Algebra, matrix multiplication, as an illustration for developing a CAL stream kernel and optimizing it to get the best possible performance from the CAL device. It implements multiplication of two 2-dimensional matrices using CAL; it then demonstrates performance optimizations to achieve an order-of-magnitude performance improvement.

3.5.1 Problem Description

If A is an m-by-k matrix, and B is an k-by-n matrix, their product is an m x n matrix denoted by AB. The elements of the product matrix AB are given by:

$$(AB)_{ij} = \sum_{r=1}^{k} a_{ir}b_{rj} = a_{i1}b1j + a_{i2}b_{2j} + ... + a_{ik}b_{kj}$$

for each pair (i, j) in $1 \le i \le m$ and $1 \le j \le k$. Figure 3.8 shows this operation for a single element in the output matrix C.

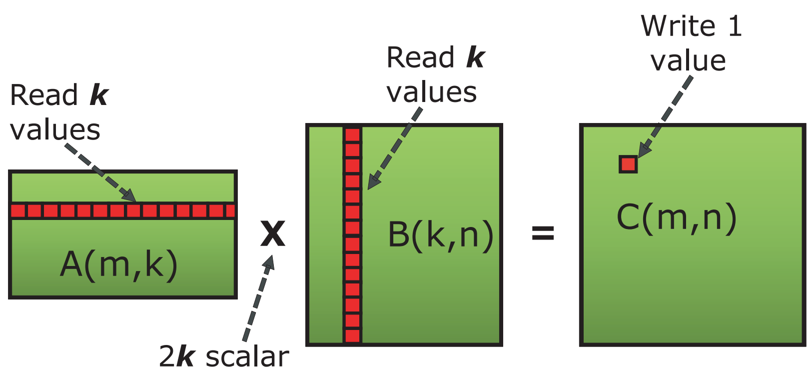

Figure 3.8 Multiplication of Two Matrices

3.5.2 Basic Implementation

It is easy to see from Figure 3.8 that the complete operation involves mkn multiplications and mkn additions. Thus, the complexity of the algorithm is $O(n^3)$. Notice that the computation of each element in the output matrix requires k values to be read, each from matrices A and B, followed by 2k scalar operations (k additions and k multiplications).

The following code contains the pseudo-code for the basic matrix-matrix multiplication algorithm that can be implemented on a CAL device.

```
/////////////////////////////////////////////////////////////////
// (i,j) is the index of the current element being processed
/////////////////////////////////////////////////////////////////
input A; // Input matrix (m, k) in size
input B; // Input matrix (k, n) in size
output C; // Output matrix (m, n) in size

void main()
{
       // Initialize the element to zero
       C[i,j] = 0.0;
```

```
    // Iterate over i'th row in matrix A and j'th column in matrix B
    // to compute the value of C[i,j]
    for (p=0; p<k; p++)
        C[i,j] += A[i,p] * B[p,j];
}
```

The output domain is the output buffer C, which is m-by-n in size. The same code is executed for each element in this domain to compute, and write to, individual elements in the output matrix.

The performance of the above algorithm is not optimal because of the poor cache hit ratio while accessing the elements in input matrices. The stream kernel accesses elements along a given column (j) of matrix B for each element in the output matrix. Assuming that memory in the input buffers is arranged in row-major order, and assuming that the size of each cache block is smaller than the row size, n, successive memory reads from matrix B come from different cache blocks. Further assuming that matrix B is bigger than the size of the cache, each memory read might result in a cache miss. Usually, however, some data reuse occurs since adjacent elements in the matrix are processed by the other element processors in the device. Also, on stream processors, the internal memory layout uses tiling, which further improves the data reuse.

3.5.3 Optimized Implementation

One commonly used algorithm for improving the cache hit ratio performs the following operations:

- Divide the input and output matrices into sub-matrices.
- Compute the product matrix one block at a time, by multiplying blocks from the input matrices.

It has been shown that the matrix multiplication operation can also be written in blocked form by dividing matrix A in MxK blocks and matrix B in KxN blocks. The resulting matrix, C, has MxN blocks. Figure 3.9 shows this decomposition. Elements of output matrix C are computed block-by-block, by multiplying blocks from matrices A and B given by the following equation.

$$c_{ij} = \sum_{r=1}^{K} a_{ir} b_{rj}$$

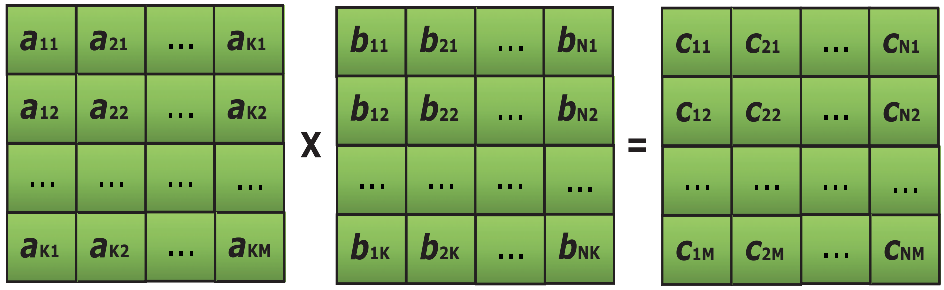

Figure 3.9 Blocked Matrix Multiplication

The modified block multiplication algorithm results in much better cache hits compared to the original algorithm. To understand this better, assume:

- the size of the sub-blocks in matrices A and B are chosen to be the same as the size of a cache block, s, used by the device
- the stream processor has separate caches for memory read and write operations,
- the total size of the read cache is ≥ 4s (the size of 4 cache blocks.)

Now, for a given block in output matrix C, there is only one cache miss per block. Subsequent memory reads are serviced from the cache.

The following code shows the pseudo-code for the modified block algorithm. This implementation adds further optimizations to the general block algorithm discussed above.

```
///////////////////////////////////////////////////////////////
// (i,j) is the index of the current fragment
///////////////////////////////////////////////////////////////
input A00, A01, A10, A11; // Input matrices (m/4, k/4) in size, 4-values per element
input B00, B01, B10, B11; // Input matrices (k/4, n/4) in size, 4-values per element
output C00, C01, C10, C11;// Output matrices (m/4, n/4) in size, 4-values per element

main() {

    // Initialize the elements to zero
    C00[i,j] = C01[i,j] = C10[i,j] = C00[i,j] = 0;

    // Iterate over i'th row in matrix A and j’th column in matrix B
    // to compute the values of C00[i,j], C01[i,j], C10[i,j] and C11[i,j]
    for (p = 0; p < k/4; p++)
    {
            C00[i,j].xyzw += A00[i,p].xxzz * B00[p,j].xyxy + A10[i,p].yyww * B01[p,j].zwzw;
            C10[i,j].xyzw += A00[i,p].xxzz * B10[p,j].xyxy + A10[i,p].yyww * B11[p,j].zwzw;
            C01[i,j].xyzw += A01[i,p].xxzz * B00[p,j].xyxy + A11[i,p].yyww * B01[p,j].zwzw;
            C11[i,j].xyzw += A01[i,p].xxzz * B10[p,j].xyxy + A11[i,p].yyww * B11[p,j].zwzw;
    }
}
```

Note the following important points about the stream kernel in the above implementation:

- It processes all four blocks in output matrix C within the computational loop.
- It leverages the superscalar floating units available on the SIMD engine by packing the input matrices so that each element in the input and output matrices contains four values.
  - The size of each matrix block now becomes 1/16th of the original matrix size (divided into 4 blocks with 4 values per element).
  - The number of output values computed and written by each stream kernel is 16.
  - To get the correct result, the input data must be preprocessed so that each four-component element in the input matrices contain a 2x2 micro-tile of data values from the original matrix (see Figure 3.10).
  - The matrix multiplication done inside the loop computes a 2x2 *micro-tile* in the output matrix and writes it as a four-component element. Thus, the output data also must be post-processed to re-arrange the data in the correct order.

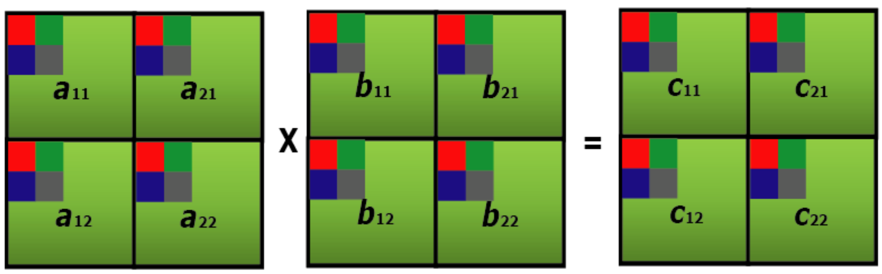

Figure 3.10 Micro-Tiled Blocked Matrix Multiplication

If the conditions specified earlier in this section hold true, the above algorithm gives near optimal performance with close to 100% cache hit ratio. However, in actual implementations, the total working set for each block multiplication might not fit in the cache. The reads cause cache misses, reducing the performance of the operation.

Note that the exact blocked decomposition scheme (values for M, N and K mentioned above) used in the implementation depend on the capabilities of the underlying stream processor architecture. For a stream processor that has a maximum of eight output buffers, the maximum number of tiles in the decomposed matrix is limited to 8x1. The best-performing algorithm that ships with the CAL SDK uses M = 8, K = 4, N = 8. With the four-component packing, it performs multiplication of 8x1 four-component blocks for matrix A with 4x1 four-component blocks of matrix B to compute 8x1 four-component blocks of matrix C.

3.6 CAL/Direct3D Interoperability

CAL features extensions providing interoperability with Direct3D 9 and Direct3D 10 on Windows XP and Windows Vista. When interoperability is used, Direct3D memory allocations can be used as inputs to, or outputs of, CAL kernels. The application must synchronize accesses of the memory from the CAL and Direct3D APIs. This can be done by using `calCtxIsEventDone` and `Direct3D` queries.

To use the interoperability, first the appropriate calD3DAssociate call must be made. This associates a CAL device to the corresponding Direct3D device. Once the devices have been associated, use the calD3DMap functions to create a CALresource from a Direct3D object. The CALresources returned from these calls can be used like any other CAL resource. When the application is finished using the allocation, it can be freed with the standard calResFree call. The CALresource must be freed before the Direct3D object is released.

Section B.4.2, “Interoperability Extensions,” page B-20, provides details of the interoperability extensions.

3.7 Advanced Topics

This section covers some advanced topics for developers who want to add new features to CAL applications or use specific features in certain AMD processors.

3.7.1 Thread-Safety

Most computationally expensive applications use multiple CPU threads to improve application performance and/or responsiveness. This typically is done by using techniques like task partitioning and pipelining in conjunction with asynchronous parallel execution on multiple processing units. In general, the CAL API is not re-entrant; that is, if more than one thread is active within a CAL function, the function invocation is not thread-safe. To invoke the same CAL function from multiple threads, the application must serialize access to these functions using synchronization primitives such as locks. The `calCtx*` functions are the exception to this rule. These functions are inherently thread safe if each thread uses a separate context. Such a model permits actions on a given context to be completely asynchronous from those on other contexts by using separate threads.

When using the CAL API in multi-threaded applications:

- CAL Compiler routines are not thread-safe. Applications invoking compiler routines from multiple threads must do proper synchronization to serialize the invocation of these routines.
- CAL Runtime routines that are either context-specific or device-specific are thread-safe. All other CAL runtime routines are not thread-safe.
- If the same context is shared among multiple threads, invocation of the `calCtx*` functions must be serialized by the application.

3.7.2 Multiple Stream Processors

Modern PC architecture allows deploying multiple PCIe devices on a system. CAL allows applications to improve performance by leveraging the computational power of multiple stream processor units that might be available on the system. Multiple devices can run in parallel by using separate threads managing each of the stream processors using one context per device[1]. CAL detects all available stream processors on the system during initialization in `calInit`. Subsequently, applications can query the number of devices on the system using `calDeviceGetCount` and then implement task partitioning and scheduling on the available devices.

Figure 3.11 shows a simple application control flow for an application using two stream processors. In this example, the main application thread sets up the application data and compiles the various CAL stream kernels. It then creates two CPU threads from the host application: one for managing each stream processor. Each of these threads internally open a CAL device, create a context on this device, and then run stream kernels. This scheme allows each of the devices to run in parallel, asynchronous to each other. The actual data or task partitioning algorithm used to load-balance the work-load between the devices is dependent on the application.

Note that CAL compiler routines are not thread safe; thus, they are called from the application thread. If the application must call compiler routines from the compute threads, it must enforce serial execution using appropriate synchronization primitives. Also, the term Stream Processor Compute Thread in Figure 3.11 is used for application-created threads that are created on the CPU and are used to manage the communication with individual stream processors. Do not confuse the term with the actual computational threads that run on the stream processor.

1. Note that the application determines whether to use a separate host CPU thread per stream processor context, or if a single host thread manages several different stream processor contexts.

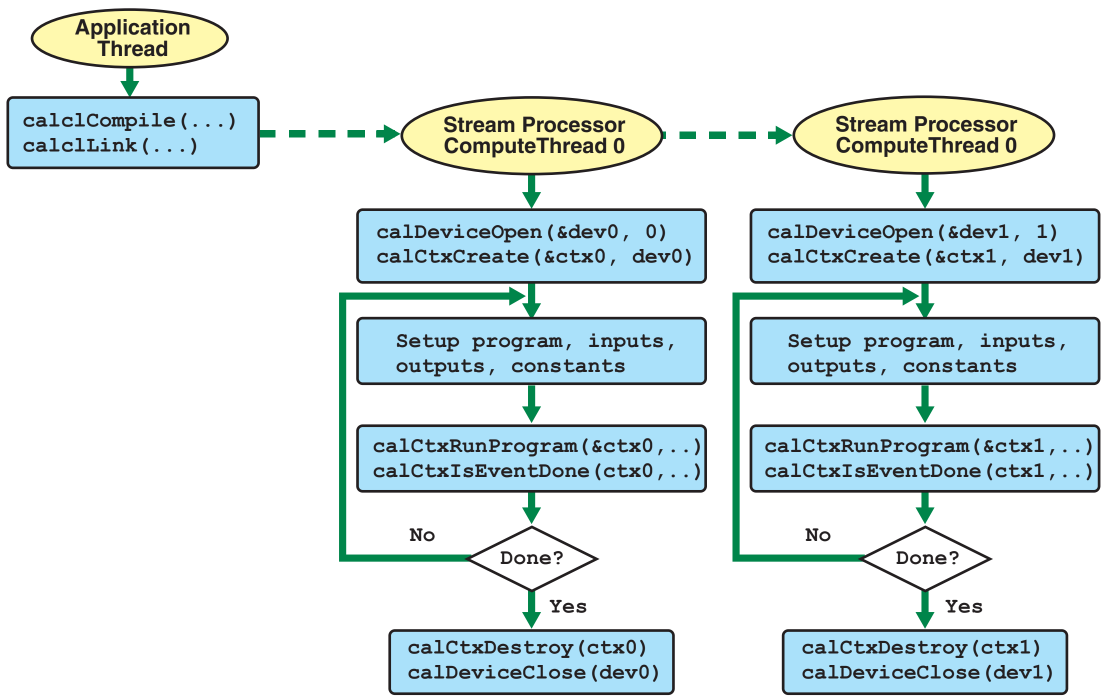

Figure 3.11 CAL Application using Multiple Stream Processors

3.7.3 Using the Global Buffer in CAL

The global buffer lets applications read from, and write to, arbitrary locations in input buffers and output buffers, respectively (see the `scatter_IL` and `gather_IL` sample programs in the `$(CALROOT)/samples/languages/IL` directory). To use global buffers, the application must perform two main modifications to a CAL application:

- request the CAL runtime to allocate global buffers when allocating resources using `CAL_RESALLOC_GLOBAL_BUFFER`, and
- specify the output (input) position for the output (input) value to be written to (read from) the global output (input) buffer.

3.7.3.1 Global Buffer Allocation

A global buffer can be allocated using the CAL runtime API: simply pass the `CAL_RESALLOC_GLOBAL_BUFFER` flag while allocating CAL resources. Global buffers can be allocated as local (stream processor) and as remote (system) memory. The following code shows this:

```
CALresource remoteGlobalRes = 0, localGlobalRes = 0;
CALformat format = CAL_FORMAT_FLOAT_1;
CALresallocflags flag = CAL_RESALLOC_GLOBAL_BUFFER;

// Allocate 2D global remote resource
calResAllocRemote2D(&remoteGlobalRes, &device, 1, width, height,
                     format, flag);
if(!remoteGlobalRes)
{
   fprintf(stdout, "Global remote resource not available on device \n");
   return -1;
}

// Allocate 2D global local resource
calResAllocLocal2D(&localGlobalRes, device, width, height, format, flag);
if(!localGlobalRes)
{
   fprintf(stdout, "Global local resource not available on device \n");
   return -1;
}
```

The rest of the mechanism for binding the resources to CPU pointers, CAL context-specific memory handles, and stream kernel inputs and outputs remain the same as normal CAL data buffers.

3.7.3.2 Accessing the Global Buffer From a Stream Kernel

The following AMD IL kernel reads data from an input buffer and uses this value as an address to write into the global output buffer. The value written is the position in the domain corresponding to the current instance of the stream kernel.

```
"il_ps_2_0\n"

// Declarations for inputs and outputs
"dcl_input_position_interp(linear_noperspective) v0\n"
"dcl_output_generic o0\n"
"dcl_cb cb0[1]\n"
"dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)\n"

// Read from (x,y)
"sample_resource(0)_sampler(0) r0, vWinCoord0.xyxx\n"

// Compute output address by computing offset in global buffer
"mad r0.x, r0.y, cb0[0].x, r0.x\n"

// Convert address from float to integer
"ftoi r1.x, r0.x\n"

// Output current position to output address in the global buffer
"mov g[r1.x], vWinCoord0.xy\n"

"ret_dyn\n"
"end\n";
```

Note that in this code:

- The global buffer is accessed using the global memory register, `g[address]`.
- The address passed to the global buffer must be a scalar integer value. The address can be a literal constant (for example, `g[2]`) or a temporary register (`r1.x` in the above example).

3.7.4 Double Precision Arithmetic

Double precision arithmetic allows applications to minimize computational inaccuracies that can result due to the use of single-precision arithmetic. Support for double precision is a crucial factor for certain applications, including engineering analysis, scientific simulations, etc. The AMD IL provides special instructions that allow applications to perform computations using 64-bit double precision in the stream processor (see the `DoublePrecision` tutorial program, located in `$CALROOT\samples\tutorial\`). Typically, double precision instructions are simply specified by prefixing the single-precision floating point instructions with d (for example, the double precision counterpart for the add instruction is dadd). For a complete reference on AMD IL syntax, as well as a list of double precision instructions, see the *AMD Intermediate Language (IL) Compiler Reference Manual*.

Assume temporary 32-bit registers. To represent 64-bit arithmetic values, two register components are used. The `f2d` and `d2f` instructions can convert from single-precision to double-precision and back. The following AMD IL kernel snippet converts two 32-bit floating values to 64-bit double precision and multiplies the values using 64-bit instructions. (Note that using conversion functions that are not in the range specified in Section 6.3 of the *AMD Compute Abstraction Layer (CAL) Intermediate Language (IL) Reference Manual* can result in the degradation of accuracy.)

```
//Convert to double precision values
"f2d r1.xy, r0.x\n"
"f2d r1.zw, r0.y\n"

// Perform double precision multiplication
"dmul r2.zw, r1.zw, r1.xy\n"
```

The `dmul` instruction performs a single double-precision multiplication using two components of the source and destination registers. Note that the following operation for double-precision multiplication also performs a single scalar multiplication operation and not a vector multiplication, as might be expected.

```
// The following operation is the same as dmul r2.xy, r1.xy, r1.xy
dmul r2, r1, r1
```

Appendix A
Brook+ Specification

This chapter describes the Brook+ language as implemented for AMD stream processors. Brook+ provides a rapid prototyping tool for developers of high-performance applications to test ideas on stream processor, multi-stream-processors, or multi-core CPU platforms.

A.1 The Structure of a Brook+ Program

Conventional C code describes a single thread of execution. Although extensions exist at the library level to manipulate threads and processes, the language specification (including the standard library) does not address parallelism.

Brook+ is an extension of C that supports an explicit model of parallelism. As explained below, it is based on a graph consisting of nodes that manipulate data and arcs that indicate the flow of data through the system (see Figure A.1 and Figure A.2). (Note that this assumes a much more regular and bounded flow of data than is the case for a traditional dataflow machine.)

A node can either restructure data or perform computations, but not both. Nodes that restructure data are called *stream operators*; nodes that perform computations are known as kernels. Both are independent processes that share a state only with that part of the system to which they are explicitly connected. A node starts when the program containing it starts; it executes whenever input data and output buffers are available; it ends when its parent program has completed execution.

An arc, known as a *stream*[1] in Brook+, connects two nodes. It does not provide any storage; instead, it maps the output of one node to the input(s) of one or more other nodes. (Implementations are permitted to introduce intermediate storage for streams, and often do, so long as this storage is transparent to the code.)

Brook+ also provides iterators that linearly interpolate values across a stream. These are like kernels that take no inputs and compute a trivial function of the stream indexes.

1. Streams provide connectivity between processing stages. A stream is a reference to an N-dimensional array of identically-typed primitive elements (a container with a coordinate space); however, it has more restricted access semantics than do conventional arrays. These restrictions permit optimization of both storage requirements and computation locality, providing higher performance for those algorithms that this model can accommodate.

The symbols shown in Figure A.1 represent the basic building blocks described above.

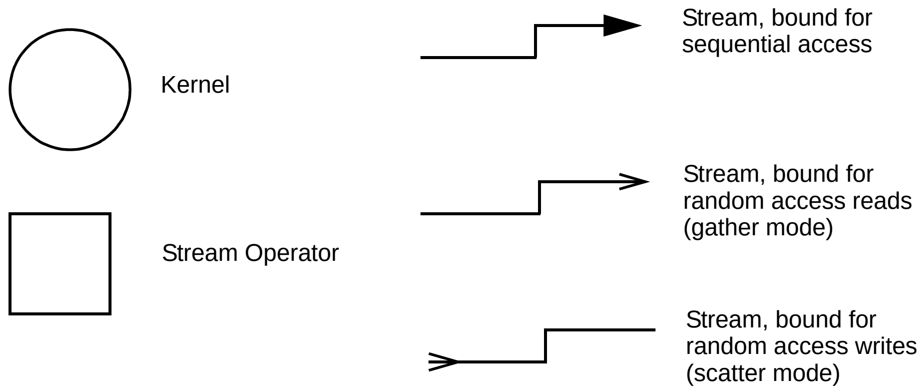

Figure A.1 Symbols for Brook+ Building Blocks

The simple illustration in Figure A.2 gives a context for these symbols. It represents a multiply-accumulate operation applied to a 10x20 grid of points.

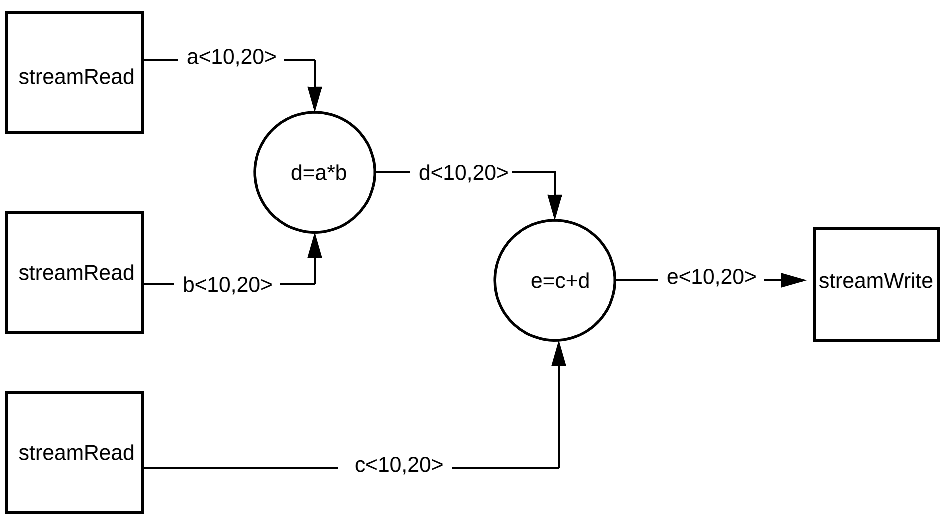

Figure A.2 Simple Streamed Multiply-Add

A.2 Primitive Data Types

These types can be used as primitive elements.

`int`	32-bit integer, signed by default
`float`	32-bit floating point
`double`	64-bit floating point; this can have a maximum of two elements

These primitive types can be aggregated using `struct` sub-scripting to generate more complex types of stream elements.

For example:

```
struct five_floats
{
   float4 a;
   float b;
};
```

is a valid Brook+ data type.

Brook+ provides built-in short vector types for `float`, `double`, and `int`; this lets code be tuned explicitly for commonly available short-SIMD machines. Here, short vector means 2 to 4 elements long. The names of these types are built from the name of their base type, with the size appended as a suffix (for example: “`float3`”, and “`int2`”). These short-vector forms also can be used as primitive elements.

Access to the fields of a short vector type is through structure member syntax, as in standard C code. For example, the float short vectors have the following equivalence:

```
float2 = struct {floatx; floaty}
float3 = struct {floatx; floaty; floatz}
float4 = struct {floatx; floaty; floatz; floatw}
```

When an operator is applied to operands of a short vector type, it is equivalent to applying the operator to each field individually. For example:

```
float2 a, b, c, d;
d = (a>b)?(a>c?a:c):(b>c?b:c);
```

is equivalent to:

```
float2 a, b, c, d;
d.x = (a.x>b.x)?(a.x>c.x?a.x:c.x):(b.x>c.x?b.x:c.x);
d.y = (a.y>b.y)?(a.y>c.y?a.y:c.y):(b.y>c.y?b.y:c.y);
```

(In other words, comparison operators produce an implicit short vector Boolean type.)

A.3 Streams and Stream Operators

This section describes the function of streams, the syntax for stream declarations, and how to use stream operators.

A.3.1 Streams

Streams provide connectivity between processing stages. A stream is a *reference* to an N-dimensional array of identically-typed primitive elements (a container with a coordinate space); however, it has more restricted access semantics than do conventional arrays. These restrictions permit optimization of both storage requirements and computation locality, providing higher performance for those algorithms that this model can accommodate.

Logically, streams do not cause storage to be allocated; however, implementations often allocate large amounts of intermediate storage to contain the data flowing around the system in streams.

As with C arrays, all dimensions but the left-most (slowest changing) must have explicitly specified bounds. The uppermost dimension can be specified implicitly.

A.3.2 Stream Declarations

The syntax for specifying a stream is similar to other C variable or type declarations, except that angle brackets are used to mark the type/variable as a stream and to delineate the stream dimensions. For example:

`float a<>;`	1D, unspecified length containing `float` elements.
`float b<>[2][3];`	1D, unspecified length containing `float[2][3]` elements.
`int c<100>;`	1D, 100 `int` elements long.
`int d<100,200,300>;`	3D, 100x200x300 `int` elements in size.
`double e<,100>;`	2D, unspecified length but 100 `double` elements wide.

Unspecified lengths are permitted only for declarations that form part of formal parameters[1], all other declarations must specify all sizes explicitly. All dimensions must be integer expressions.

The elements of a stream cannot be accessed from regular C code; they are visible only to kernels and stream operators. (See Section A.3.3.1, “I/O Stream Operators,” page A-5, for more details.)

Streams can contain arrays as primitive elements, but arrays of streams are not permitted.

The current implementation supports streams containing up to 2^{23} elements.

A.3.3 Stream Operators

A stream operator looks like a function call and either:

- remaps a stream, or presents a remapped view of a stream, without changing data at the element level, or
- provides an I/O mechanism between the streaming world of the Brook+ code and the enclosing host environment.

1. Formal parameters are the names given in the function definition; this is distinct from actual parameters, which are the values passed to the function.

A.3.3.1 I/O Stream Operators

The following subsections describe copying data to, and from, system memory.

Copying Data from System Memory –

The code:

streamRead(*destination_stream*, *source_array*)

copies the elements of source_array to destination_stream.

The number of dimensions, size, and element types must match; otherwise, the behavior is undefined.

Copying Data to System Memory –

The code:

streamWrite(*source_stream*, *destination_array*)

copies elements from source_stream to destination_array.

The number of dimensions, size, and element types must match,; otherwise, the behavior is undefined.

A.3.3.2 Implicit Insertion of Stream Operators

If a kernel is bound to a stream the size of which is different from that specified in the kernel's formal parameter, Brook+ Beta-1 automatically inserts an implicit stream operator that rescales the stream to match. The following examples illustrate this.

The first example is an instance of downscaling from a larger stream to a smaller one.

```
#include <stdio.h>

kernel void copy(float a<>, out float b<>)
{
    b = a;
}

int main(int argc, char **argv)
{
    float src<10>;
    float dst<5>;
    float s[10];
    float d[5];
    int i;

    for (i = 0; i < 10; i++)
    {
        s[i] = (float)i;
    }
```

```
    streamRead(src, s);
    copy(src, dst);
    streamWrite(dst, d);

    for (i = 0; i < 10; i++)
    {
        printf("%4.1f ", s[i]);
        if (i < 5)
        {
            printf("%4.1f", d[i]);
        }
        puts("");
    }
}
```

Here, the source stream is twice the size of the destination stream; so the kernel downscales during the copy process by skipping every second element in the input. The result of running this example is:

```
0.0  0.0
1.0  2.0
2.0  4.0
3.0  6.0
4.0  8.0
5.0
6.0
7.0
8.0
9.0
```

Upscaling is similar:

```
#include <stdio.h>
kernel void copy(float a<>, out float b<>)
{
    b = a;
}
int main(int argc, char **argv)
{
    float src<5>;
    float dst<10>;
    float s[5];
    float d[10];
    int i;
    for (i = 0; i < 5; i++)
    {
        s[i] = (float)i;
    }
    streamRead(src, s);
    copy(src, dst);
    streamWrite(dst, d);
    for (i = 0; i < 10; i++)
    {
        if (i < 5)
        {
            printf("%4.1f ", s[i]);
        }
        else
```

```
            {
                printf("     ");
            }
            printf("%4.1f\n", d[i]);
        }
    }
```

Here, the situation is reversed, and the kernel upscales the input stream by replicating each element. The result is:

```
 0.0  0.0
 1.0  0.0
 2.0  1.0
 3.0  1.0
 4.0  2.0
      2.0
      3.0
      3.0
      4.0
      4.0
```

A.4 Kernels

Kernels are the part of the streaming model used to define computation. The most basic form is simply mapped over input data and produces one output item for each input tuple. Subsequent extensions of the basic model provide random-access functionality, variable output counts, and reduction/accumulation operations.

A.4.1 Kernel Types

There are two kernel types: basic and reduction. The following subsections provide information about each.

A.4.1.1 The Basic Kernel

The simplest type of kernel takes an element from the same location in each input stream, computes a function of it, then writes it to the corresponding location in the output stream. This is repeated for every element.

```
void kernel mad(float a<>, float b<>, float c, out float d<>)
{
   d = a * b + c;
}
```

The input streams must all be of the same size for this operation to be meaningful (however, see Section A.3.3.2, “Implicit Insertion of Stream Operators,” page A-5). When the sizes can be determined at compile-time, implementations are required to check correctness. When the stream sizes cannot be determined at compile-time, provide a compile-time option to enable or disable runtime checking.

The current implementation supports binding 128 inputs and 8 outputs to a single kernel.

A.4.1.2 Reduction Kernels

Reductions are kernels that decrease the dimensionality of a stream by folding along one axis using an associative and commutative binary operation. The requirement that the operation be associative and commutative means that the result is independent of evaluation order, modulo, any issues due to limited floating point precision.

Brook+ provides two mechanisms for specifying reductions: *reduction variables* and *reduction functions*.

A reduction variable is specified as part of a kernel and operated on using any of the C assignment operators that satisfies the associativity and commutativity requirements; that is: +=, *=, |=, and ^=.

Reduction variables can be any of the primitive types specified above.

For example:

```
void kernel sum(float a<>, reduce float b)
{
   b += a;
}
```

Reduction variables do not necessarily have to be updated for every kernel invocation.

For example:

```
void kernel cond_sum(float a<>, reduce float c)
{
   if (a > 10.0)
   {
      c += a;
   }
}
```

Provide the correct identities (0 for addition, 1 for multiplication, ∞ for max, etc.) as part of the invocation of the reduction.

In addition to the associative assignment operators listed above, the programmer also can specify a *reduction function* that is guaranteed to meet the same requirements. (This is not checked by the compiler). A reduction function is marked by prefixing the function definition and the reduction variable with the `reduce` keyword[1].

1. Currently, prefixing the variable is sufficient to mark the kernel as a reduce kernel.

For example:

```
reduce void max_reduce(double a, reduce double b)
{
   if (a > b)
      b = a;
}

reduce void min_reduce(double a, reduce double b)
{
   if (a < b)
      b = a;
}
```

It can be called either as a kernel from the host code, or used as a subkernel by an enclosing kernel (which can itself be a reduction kernel).

A.4.1.3 Partial Reductions

A partial reduction is possible if the target stream has the same number of dimensions as the source stream. This reduces size but not dimensionality. Each dimension of the source must be: (a) no smaller than the corresponding dimension of the target, and (b) an integer multiple of the corresponding dimension of the target.

For example, assuming a reduction kernel called `sum()`:

```
float s<100,200>;
float t<100>;
sum(s, t);
float u<100,50>;
sum(s, u);
```

Each element of `t` is generated by summing a 1x200 strip from `s`, and each element of `u` is generated by summing a 1x4 strip from `s`.

A.4.2 Kernel-Specified Communication Patterns

Brook+ is based on a separation of communication and computation, with stream operators defining communication patterns and kernels defining computation. Some users find this too restrictive, so a mechanism has been provided to allow kernels to specify their own communication patterns.

If a stream is bound to a kernel using array brackets rather than stream brackets, the code inside the kernel can access any of the elements of the stream, not just the single element to which the kernel is mapped. This is very similar to a C array operation, except that the index is presented as a float2 (rather than 2 floats in C).

For example:

```
kernel void gather_ex_1(float2 a<>, float b[100][100], out float c<>)
{
   c = b[a];
}
```

Indices can be pulled directly from a stream, or computed as part of the kernel operation:

```
kernel void indexing(float3 a<>, float b[100][100][100], out float c<>)
{
   float3 d = some_function(a);
   c = b[d];
}
```

A stream must be bound write-only or read-only. Read-write binding is not permitted.

Note that specifying communication patterns inside kernels rather than using stream operators can degrade performance.

A.4.3 Calling Other Code from Kernel Code

Kernels can call other functions defined in the same `.br` file or any files it includes; however, there are restrictions.

- A top-level kernel must have a return type of `void` to be callable from host code. Subkernels can return data of any non-stream type. A subkernel also can be bound to streams propagated from its parent kernel.
- Subkernels are logically expanded inline, so recursion is not permitted.
- Kernels cannot call stream operators.

A.4.4 Restrictions on Kernel Code

Kernels can use both stream and non-stream parameters as inputs. Generally, only streams can be used as outputs (but see reductions, below).

Within a kernel definition, the following restrictions apply:

- The `goto`, `volatile`, and `static` keywords are prohibited.
- All variables must be of automatic storage class (that is, declared on the stack).
- Pointers are not supported.
- Recursion is not allowed.
- Precise exceptions are not supported.
- Any pointers passed into Brook+ code are required not to alias each other.
- Brook+ functions callable from C code are required to fully specify the sizes of array arguments.
- Storage allocated by Brook+ code can not be accessed by external code except during the lifetime of external functions called from that Brook+ code; and streams are never accessible to non-Brook+ code.

A Brook+ project can be made up of both C/C++ and Brook+ source files, with the Brook+ files having the extension `.br`. Within a Brook+ file, the following restrictions apply:

- Brook+ functions can not call functions declared in C files.
- Preprocessor directives are passed through to the host C++ compiler untouched and uninterpreted.

A.5 Standard Library Functions and Intrinsics

The following is a listing and description of the kernel intrinsics.

indexof()	The indexof operator is applied to a stream and returns a `float` (or `floatN`) type containing the index of the element that the kernel currently being mapped over. This operator is not valid for reduction or gather streams.
abs(x)	Absolute value of x.
acos(x)	Inverse cosine of x.
asin(x)	Inverse sine of x.
clamp(x,a,b)	Clamps the supplied value to be between an upper and lower limit. $a \le clamp(x) \le b$.
cos(x)	Cosine of x.
cross(x,y)	Cross product of the two vectors x and y.
dot(x,y)	Dot product of the two vectors x and y.
exp(x)	e^x
floor(x)	$\lfloor x \rfloor$
fmod(x,y)	Returns f such that $x = i * y + f$, where i is an integer, f has the same sign as x and $\lvert f \rvert < \lvert y \rvert$.
frac(x)	Returns the fractional part of x.
isfinite(x)	Returns true if x is finite, false (0) otherwise.
isinf(x)	Returns true if x is infinite, false (0) otherwise.
isnan(x)	Returns true if x is NaN, false (0) otherwise.
lerp(x,y,a)	$(1 - a)x + ay;\ 0 \le a \le 1$
log(x)	$\ln(x)$
max(x,y)	Returns the greater of x or y.
min(x,y)	Returns the lesser of x or y.
normalize(x)	Normalizes a vector, returning $\frac{x}{\lvert x \rvert}$.
pow(x,y)	x^y
rsqrt(x)	$\frac{1}{\sqrt{x}}$
round(x)	Rounds x to the nearest integer by adding 0.5 and truncating.

sign(x)	Returns the sign of *x*, if x is 0 then sign(*x*) is also 0.
sin(x)	Sine of *x*.
sqrt	$\sqrt{x}$

Appendix B
The AMD Compute Abstraction Layer (CAL) API Specification

The AMD Compute Abstraction Layer (CAL) provides a forward-compatible, interface to the high-performance, floating-point, parallel processor arrays found in AMD stream processors and CPUs.

The CAL API is designed so that:

- the computational model is processor independent.
- the user can easily switch from directing a computation from stream processor to CPU or vice versa.
- it permits a dynamic load balancer to be written on top of CAL.
- CAL is a lightweight implementation that facilitates a compute platform such as Brook+ to be developed on top of it.

CAL is supported on R6xx and newer generation of AMD graphics processors and all CPU processors. It runs on XP, Vista, Linux, as well as both 32-bit and 64-bit versions.

B.1 Programming Model

The CAL application executes on the CPU, driving one or more stream processors. A stream processor is connected to two types of memory: local (stream processor) and remote (system). Contexts on a stream processor can read and write to both memory pools. Context reads and writes to local memory are faster than those to remote memory. The master process also can read and write to local and remote memory. Typically, the master process has higher read and write speeds to the remote (system) memory of the stream processors. The master process submits commands or jobs for execution on the multiple contexts of a stream processor. The master process also can query the context for the status of the completion of these tasks. Figure B.1 illustrates a CAL system.

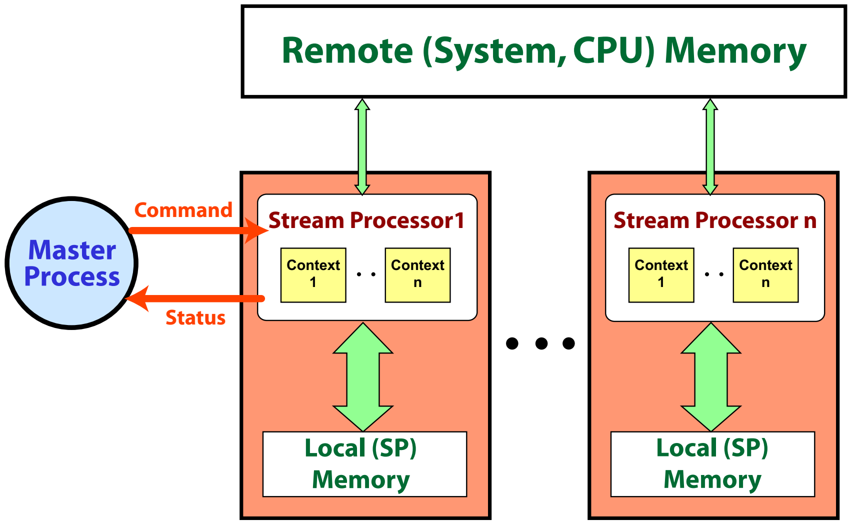

Figure B.1 CAL System

A stream processor has a one or more SIMD engines. The computational function (kernel) is executed on these arrays. Unlike CPUs, stream processors contain a large array of SIMD processors. The inputs and outputs to the kernel can be set up to reside either in the local or the remote memory. A kernel is invoked by setting up one or more outputs and specifying a domain of execution[1] for this output that must be computed. In the case of a stream processor having multiple processors (such as a stream processor), a scheduler distributes the workload to various SIMD engines on the stream processor.

The CAL abstraction divides commands into two key types: device and context. A device is a physical stream processor visible to the CAL API. The device commands primarily involve resource allocation (local or remote memory). A context is a queue of commands that are sent to a stream processor. There can be parallel queues for different parts of the stream processor. Resources are created on stream processors and are mapped into contexts. Resources must be mapped into a context to provide scoping and access control from within a command queue. Each context represents a unique queue. Each queue operates independently of each other. The context commands queue their actions in the supplied context. The stream processor does not execute the commands until the queue is flushed. Queue flushing occurs implicitly when the queue is full or explicitly through CAL API calls.

Resources are accessible through multiple contexts on the same stream processor and represent the same underlying memory (Figure B.2). Data sharing across contexts is possible by mapping the same resource into multiple contexts. Synchronization of multiple contexts is the client's responsibility.

1. A specified rectangular region of the output buffer to which threads are mapped.

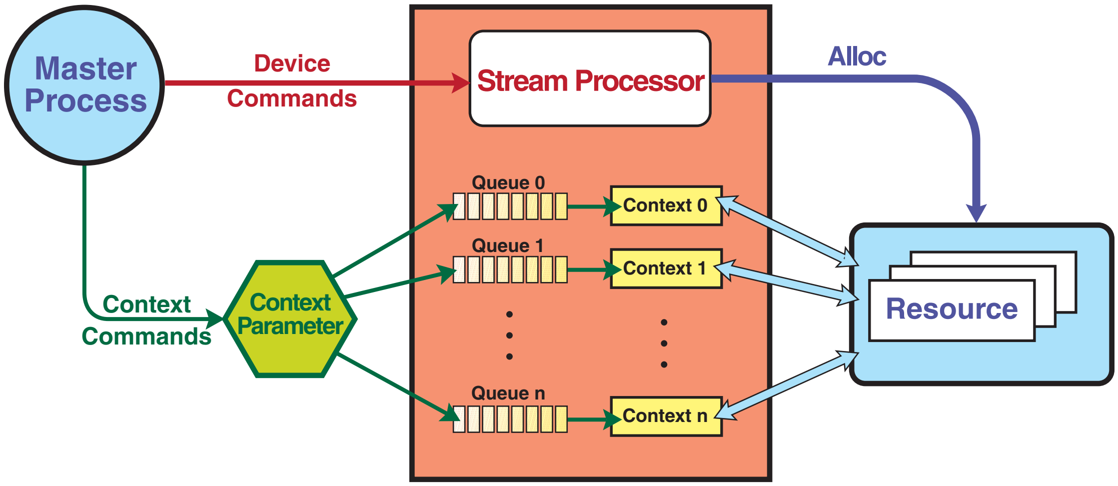

Figure B.2 Context Queues

B.2 Runtime

The CAL runtime comprises the system, stream processor management, context management, memory management, program loader, computational component, and synchronization component. The following subsections describe these.

B.2.1 System

The system component initializes and shuts down a CAL system. It also contains methods to query the version of the CAL runtime. Section B.3.1, "System Component," describes the relevant API.

B.2.2 Device Management

A machine can have multiple processing units. Each of these is known as a device. The device management component opens and closes a device; it also queries the devices and their attributes. Section B.3.2, "Device Management," describes the relevant API.

B.2.3 Memory Management

The memory management component allocates and frees memory resources. These can be local or remote to a processing device. Memory resources are not directly addressed by contexts; instead, they create memory handles from a memory resource for any specific context. This allows access to the same memory resource by two memory contexts through two memory handles.

The API provides function calls to map the memory handles to CPU address space for access by the master process.

Currently, shared remote resources across devices are not supported.

Section B.3.3, "Memory Management," describes the relevant API.

B.2.4 Context Management

A device can have multiple contexts active at any time. This component creates and destroys contexts on a particular device. Section B.3.4, "Context Management," describes the relevant API.

B.2.5 Program Loader

The program loader loads a CAL image onto a context of a device to generate a module. An image is generated by compiling the source code into objects, which are then linked into an image. It is possible to get handles to the entry points and names used in the program from a loaded module. These entry point and name handles are used to setting up a computation. Section B.3.5, "Loader," describes the relevant API.

B.2.6 Computation

This component sets up and executes a kernel on a context. This includes:

- setting up the memory for inputs and outputs,
- triggering a kernel.

This component also handles data movement by a context. The API provides function calls for querying if a computational task or data movement task is done. Section B.3.6, "Computation," describes the relevant API.

B.3 Platform API

The following subsections describe the APIs of the CAL runtime components.

B.3.1 System Component

The following function calls are specific to the system component of the CAL runtime.

calInit

Syntax	**CALresult calInit(void)**	
Description	Initializes the CAL API for computation.	
Results	CAL_RESULT_ERROR	Error.
	CAL_RESULT_ALREADY	CAL API has been initialized already.
	CAL_RESULT_OK	Success.
	CAL_RESULT_NOT_INITIALZIED	CAL API has not been initialized.

calShutdown

Syntax	**CALresult calShutdown(void)**	
Description	Shuts down the CAL API. Must be paired with calInit. An application can have any number of calInit - calShutdown pairs. Calling calShutdown destroys any open context, frees allocated resources, and closes all open devices.	
Results	CAL_RESULT_NOT_INITIALZIED	Any CAL call outside a calInit - calShutdown pair.

calGetVersion

Syntax

```
CALresult calGetVersion(
    CALuint* major,
    CALuint* minor,
    CALuint* imp)
```

Description	Returns the major, minor, and implementation versions numbers of the CAL API.	
Results	CAL_RESULT_OK	Success.
	CAL_RESULT_BAD_PARAMETER	Error. One or more parameters are null.

B.3.2 Device Management

The following function calls are specific to the device management component of the CAL runtime.

calDeviceGetCount

Syntax	**CALresult calDeviceGetCount(CALuint* count)**	
Description	Returns the numbers of processors available to the CAL API for use by applications.	
Results	CAL_RESULT_OK	Success.
	CAL_RESULT_ERROR	Error. Count is assigned a value of zero.

calDeviceGetAttribs

Syntax	**CALresult calDeviceGetAttribs (** **CALdeviceattribs* attribs,** **CALuint ordinal)**	
Description	Returns device-specific information about the processor in attribs. The device is specified by ordinal, which must be in the range of zero to the number of devices returned by calDeviceGetCount minus one. The device does not have to be open to obtain information about it. The struct_size field of the CALdeviceattribs structure must be filled out prior to calling calDeviceGetInfo.	
Results	CAL_RESULT_OK	Success, and attribs contains information about the device.
	CAL_RESULT_INVALID_PARAMETER	Error if ordinal is not a valid device number.
	CAL_RESULT_ERROR	Error if information about the device cannot be obtained.
		On error, the contents of attribs is undefined. See CALdeviceattribs for details on the CALdeviceattribs structure.

calDeviceOpen

Syntax	**CALresult calDeviceOpen(** **CALdevice* dev,** **CALuint ordinal)**	
Description	Opens a device indexed by ordinal. A device must be closed before it can be opened again in the same application. Always pair this call with calDeviceClose.	
Results	CAL_RESULT_OK	Success, and dev is a valid handle to the device.
	CAL_RESULT_INVALID_PARAMETER	Error if ordinal is not a valid device number.
	CAL_RESULT_ERROR	Error if information about the device cannot be opened.
		On error, dev is zero.

calDeviceGetStatus

Syntax
```
CALresult calDeviceGetStatus (
    CALdevicestatus* status,
    CALdevice dev)
```

Description Opens a device indexed by `ordinal`. A device must be closed before it can be opened again in the same application. Always pair this call with `calDeviceClose`.

Results

CAL_RESULT_OK	Success, and dev is a valid handle to the device.
CAL_RESULT_INVALID_PARAMETER	Error if `ordinal` is not a valid device number.
CAL_RESULT_ERROR	Error if information about the device cannot be opened. On error, dev is zero.

calDeviceClose

Syntax
```
CALresult calDeviceClose(CALdevice dev)
```

Description Closes a device specified by the dev handle. When a device is closed, all contexts created on the device are destroyed, and all resources on the device are freed. Always pair this call with `calDeviceOpen`.

Results

CAL_RESULT_OK	Success: dev is a valid handle to the device.
CAL_RESULT_ERROR	The overall state is assumed to be as if `calDeviceClose` was never called.

B.3.3 Memory Management

The following function calls are specific to the memory management component of the CAL runtime.

calResAllocLocal2D

Syntax	**CALresult calResAllocLocal2D(** **CALresource* res,** **CALdevice device,** **CALuint width,** **CALuint height,** **CALformat format,** **CALuint flags)**
Description	Allocates memory local to a stream processor. The device specifies the stream processor to allocate the memory. This memory is structured as a two-dimensional region of *width* and *height* with a format. The maximum dimensions are available through the calDeviceGetInfo function. The flags parameter is used to specify a basic level of use for the memory. For local memory, the value must be zero unless the memory is used for memory export. If the memory is used for memory export, then flags must be CAL_RESALLOC_GLOBAL_BUFFER.

Results	CAL_RESULT_OK	Success, and res is a handle to the memory resource.
	CAL_RESULT_BAD_HANDLE	Error if dev is not a valid device.
	CAL_RESULT_ERROR	Error if the memory can not be allocated.
		On error, res is zero.

calResAllocRemote2D

Syntax

```
CALresult calResAllocRemote2D(
    CALresource* res,
    CALdevice* sharedDevices,
    CALuint deviceCount,
    CALuint width,
    CALuint height,
    CALformat format,
    CALuint flags)
```

Description Allocates memory remote to `deviceCount` number of devices in the `sharedDevices` array. The memory is system memory, remote to all stream processors. This memory is structured as a two-dimensional region of width and height with a format. The maximum dimensions are available through the `calDeviceGetInfo` function.

The *flags* parameter specifies a basic level of use for the memory. For remote memory, zero means the memory is allocated in uncached system memory, `CAL_RESALLOC_CACHEABLE` forces the memory to be CPU cachable.

One benefit of devices being able to write to remote (system) memory is performance. For example, with large computational kernels, it sometimes is faster for the stream processor contexts to write directly to remote memory than it is to do process them in two steps: stream processor context writing to local memory, and copying data from stream processor local memory to remote system memory.

Results

`CAL_RESULT_OK`	Success, and `res` is a handle to the memory resource.
`CAL_RESULT_BAD_HANDLE`	Error if any device in `sharedDevices` is not valid.
`CAL_RESULT_ERROR`	Error if the memory can not be allocated.
	On error, `res` is zero.

calResAllocLocal1D

Syntax

```
CALresult calResAllocLocal1D(
    CALresource* res,
    CALdevice device,
    CALuint width,
    CALformat format,
    CALuint flags)
```

Description Allocates memory local to a stream processor. The device to allocate the memory is specified by device. This memory is structured as a one-dimensional region of *width* with a *format*. The maximum dimensions are available through the calDeviceGetInfo function.

The flags parameter is used to specify a basic level of use for the memory. For local memory, the value must be zero unless the memory is used for memory export. If the memory is used for memory export, flags must be CAL_RESALLOC_GLOBAL_BUFFER.

Results

CAL_RESULT_OK	Success, and res is a handle to the memory resource.
CAL_RESULT_BAD_HANDLE	Error if dev is not a valid device.
CAL_RESULT_ERROR	Error if the memory can not be allocated.
	On error, res is zero.

calResAllocRemote1D

Syntax

```
CALresult calResAllocRemote1D(
    CALresource* res,
    CALdevice* sharedDevices,
    CALuint deviceCount,
    CALuint width,
    CALformat format,
    CALuint flags)
```

Description Allocates memory remote to `deviceCount` number of devices in the `sharedDevices` array. The memory is system memory (remote to all devices). It is structured as a one-dimensional region of *width* with a *format*. The maximum dimensions are available through the `calDeviceGetInfo` function.

The `flags` parameter specifies a basic level of use for the memory. For remote memory, zero means the memory is allocated in uncached system memory, `CAL_RESALLOC_CACHEABLE` forces the memory to be CPU-cachable.

One benefit of devices being able to write to remote (system) memory is performance. For example, with large computational kernels, it sometimes is faster for the stream processor contexts to write directly to remote memory than it is to do process them in two steps: stream processor context writing to local memory, and copying data from stream processor local memory to remote system memory.

Results

`CAL_RESULT_OK`	Success, and `res` is a handle to the memory resource.
`CAL_RESULT_BAD_HANDLE`	Error if any device in `sharedDevices` is not valid.
`CAL_RESULT_ERROR`	Error if the memory can not be allocated.
	On error, `res` is zero.

calResFree

Syntax **`CALresult calResFree(CALresource res)`**

Description Releases the memory resources as specified by handle `res`.

Results

`CAL_RESULT_OK`	Success.
`CAL_RESULT_BAD_HANDLE`	Error if `res` is an invalid handle
`CAL_RESULT_BUSY`	Error if the resource is in use by a context.
	On error, the state is as if `calResFree` had never been called. Use `calCtxReleaseMem` to release a resource handle from a context.

calResMap

Syntax

```
CALresult calResMap(
    CALvoid** pPtr,
    CALuint* pitch,
    CALresource res,
    CALuint flags)
```

Description Returns a CPU-accessible pointer to the specified resource `res`. The CPU pointer address is returned in `pPtr`. For two-dimensional surfaces, the count, in the number of elements across the width, is returned in `pitch`. The `flags` field must be zero.

The CAL client must ensure the contents of the resource do not change; this is done by ensuring that all outstanding kernel programs that affect the resource are complete prior to mapping.

The `calResMap` function blocks the thread until the CPU-accessible pointer is valid. For local surfaces, this can mean the implementation performs a copy of a resource and waits until the copy is complete. For remote surfaces, a pointer to the surface is returned without copying contents.

Results

CAL_RESULT_OK	Success, and a valid CPU pointer returned in `pPtr`. Pitch is the number of elements across for each line in a two-dimensional image.
CAL_RESULT_BAD_HANDLE	Error if `res` is an invalid handle
CAL_RESULT_ERROR	Error if the surface can not be mapped.
CAL_RESULT_ALREADY	Returned if the resource is already mapped
	On error, `pPtr` and `pitch` are zero.

calResUnmap

Syntax

```
CALresult calResUnmap (CALresource res)
```

Description Releases the address returned in `calResMap`. All mapping resources are released, and CPU pointers become invalid. This must be paired with `calResMap`.

Results

CAL_RESULT_OK	Success.
CAL_RESULT_BAD_HANDLE	Error if `res` is an invalid handle
CAL_RESULT_ERROR	The resource is not mapped, and `Unmap` was called.

B.3.4 Context Management

The following function calls are specific to the context management component of the CAL runtime.

calCtxCreate

Syntax
```
CALresult calCtxCreate(
    CALcontext* ctx,
    CALdevice dev)
```

Description Creates a context on the device specified by dev. Multiple contexts can be created on a single device.

Results		
	CAL_RESULT_OK	Success, and ctx contains a handle to the context.
	CAL_RESULT_BAD_HANDLE	Error if res is an invalid handle.
	CAL_RESULT_ERROR	A context can not be created.
		On error, ctx is zero.

calCtxDestroy

Syntax
```
CALresult calCtxDestroy(CALcontext ctx)
```

Description Destroys a context specified by the ctx handle. When a context is destroyed, all currently executing kernels are completed, all modules are unloaded, and all memory is released from the context.

Pair this call with calCtxCreate.

Results		
	CAL_RESULT_OK	Success.
	CAL_RESULT_BAD_HANDLE	Error if ctx is an invalid handle
	CAL_RESULT_ERROR	A context can not be created.
		On error, ctx is zero.

calCtxGetMem

Syntax

```
CALresult calCtxGetMem(
    CALmem* mem,
    CALcontext ctx,
    CALresource res)
```

Description Maps a resource specified by `res` into the context specified by `ctx`. The memory handle is returned in `mem`. The returned memory handle's scope is relative to the supplied context. If the supplied resource is a shared remote resource, only contexts belonging to the "shared devices" argument during creation have access to this resource.

Results

CAL_RESULT_OK	Success, and `mem` contains a handle to the memory.
CAL_RESULT_BAD_HANDLE	Error if `ctx` or `res` is an invalid handle

calCtxReleaseMem

Syntax

```
CALresult calCtxReleaseMem(
    CALcontext ctx,
    CALmem mem)
```

Description Releases the memory handle specified by `mem` from the context specified by `ctx`. The resource used to create the memory handle is updated with a release notification.

Results

CAL_RESULT_OK	Success, and `mem` contains a handle to the memory.
CAL_RESULT_BAD_HANDLE	Error if `ctx` or `mem` is an invalid handle

calCtxSetMem

Syntax

```
CALresult calCtxSetMem(
    CALcontext ctx,
    CALname name,
    CALmem mem)
```

Description Associates memory with a symbol from a compiled kernel. The memory is specified by `mem`. The symbol is specified by `name`. The context where the association occurs is specified by `ctx`. To remove an association, call `calCtxSetMem` with a null memory handle. The semantics of the kernel symbol name dictate if the memory is used for input, output, constants, or memory export.

Results

CAL_RESULT_OK	Success.
CAL_RESULT_BAD_HANDLE	Error if `ctx` or `mem` is an invalid handle.

B.3.5 Loader

The following function calls are specific to the loader component of the CAL runtime.

calModuleLoad

Syntax

```
CALresult calModuleLoad(
    CALmodule* module,
    CALcontext ctx,
    CALimage image)
```

Description

Creates a module handle from a precompiled kernel binary image and loads the image on the context specified by `ctx`. The handle for the module is returned in `module`. See *CAL Image. AMD's Compute Abstraction Layer Program Binary Format Specification* for details on the format of CALimage. Multiple images can be loaded concurrently.

The CALimage passed into `calModuleLoad` must conform to the CAL multi-binary format, as specified in the *CAL Image* document. A multi-binary consists of many different encodings of the same program. The loader chooses the best match encoding to load. The order priority for the encoding that is loaded is ISA, feature matching AMD IL, base AMD IL. All AMD IL encodings go through load-time translation to the device-specific ISA prior to being loaded.

Results

CAL_RESULT_OK	Success, and `module` is a valid handle.
CAL_RESULT_BAD_HANDLE	Error if `ctx` is an invalid handle.
CAL_RESULT_INVALID_PARAMETER	Error if `module pointer` is null.
CAL_RESULT_ERROR	Error if the binary is invalid or can not be loaded.

calModuleUnload

Syntax

```
CALresult calModuleUnload(
    CALcontext ctx,
    CALmodule module)
```

Description

Unloads the module specified by the `module` handle from the context specified by `ctx`. Unloading a module disassociates all CALname handles from their assigned memory and destroys all CALname and CALfunc handles associated with the module.

Results

CAL_RESULT_OK	Success.
CAL_RESULT_BAD_HANDLE	Error if `ctx` or `module` is an invalid handle.

calModuleGetEntry

Syntax	**CALresult calModuleGetEntry(** **CALfunc* func,** **CALcontext ctx,** **CALmodule module,** **const CALchar* procName)**	
Description	Retrieves a function by name in a loaded module. The `module` parameter specifies from which loaded module the function is retrieved. The name of the function is specified by `procName`. The returned handle can be used to execute the function using `calCtxRunProgram`.	
Results	CAL_RESULT_OK	Success, and `func` is a valid handle to the function entry point.
	CAL_RESULT_BAD_HANDLE	Error if `ctx` or `module` is an invalid handle.
	CAL_RESULT_ERROR	Error if the function name is not found in the module.
		On error, `func` is zero.

calModuleGetName

Syntax	**CALresult calModuleGetName(** **CALname* name,** **CALcontext ctx,** **CALmodule module,** **const CALchar* symbolName)**	
Description	Retrieves a symbol by name in a loaded module. The `module` parameter specifies from which loaded module to retrieve the symbol. The name of the symbol is specified by `symbolName`. The returned handle can be used to associate memory with the symbol using `calCtxSetMem`. The semantic use for the name is determined by the use in the kernel program. Symbols can be used for inputs, outputs, constants, and memory exports.	
Results	CAL_RESULT_OK	Success, and `name` is a valid handle to the symbol name.
	CAL_RESULT_BAD_HANDLE	Error if `ctx` or `module` is an invalid handle.
	CAL_RESULT_ERROR	Error if the symbol name is not found in the module.
		On error, `name` is zero.

calImageRead

Syntax	**CALresult calImageRead(** **CALimage* image,** **const CALvoid* buffer,** **CALuint size)**
Description	Creates a CALimage and populates it with information from the supplied buffer.

Results	CAL_RESULT_OK	Success.
	CAL_RESULT_ERROR	Error.

B.3.6 Computation

The following function calls are specific to the computation component of the CAL runtime.

calCtxRunProgram

Syntax	**CALresult calCtxRunProgram(** **CALevent* event,** **CALcontext ctx,** **CALfunc func,** **const CALdomain* rect)**
Description	Issues a program run task to invoke the computation of the kernel identified by `func` within a region `rect` on the context `ctx`, and returns an associated event token in `event` with this task. The run program task is not scheduled for execution until `calCtxIsEventDone` is called. Completion of the run program task can be queried by the CAL client by calling `calCtxIsEventDone` within a loop.

Results	CAL_RESULT_OK	Success.
	CAL_RESULT_BAD_HANDLE	Error if `ctx` or `func` is an invalid handle.
	CAL_RESULT_ERROR	Error if any of the symbols used by `func` are invalid or if any of the resources bound to the symbols are mapped. Use `calCtxGetErrorString` for contextual information regarding any errors.

calMemCopy

Syntax

```
CALresult calMemCopy(
    CALevent* event,
    CALcontext ctx,
    CALmem srcMem,
    CALmem destMem,
    CALuint flags)
```

Description Issues a task to copy data from a source memory handle to a destination memory handle. An event is associated with this task and is returned in `event`, and completion of this event can be queried by the master process using `calCtxIsEventDone`. Data can be copied between memory handles from:

- remote system memory to device local memory,
- remote system memory to remote system memory,
- device local memory to remote system memory,
- device local memory to same device local memory,
- device local memory to a different device local memory.

The memory is copied by the context `ctx`. It can be placed in a separate queue or the primary `calCtxRunProgram` queue of context `ctx`.

Results

CAL_RESULT_OK	Success, and `event` contains the event identifier that a client can poll to query completeness.
CAL_RESULT_BAD_HANDLE	Error if `ctx`, `srcMem`, or `dstMem` is an invalid handle.
CAL_RESULT_ERROR	Error if the source and destination memory have different sizes or formats.
	On error, `event` is zero.

calCtxIsEventDone

Syntax

```
CALresult calCtxIsEventDone(
    CALcontext ctx,
    CALevent event)
```

Description This function:

Schedules an event specified by `event` for execution.

Permits a CAL client to query if an event, specified by `event`, on the context, `ctx`, has completed.

Results

CAL_RESULT_OK	The Run Program or Mem Copy associated with the event identifier has completed.
CAL_RESULT_PENDING	Returned for events that have not completed.
CAL_RESULT_BAD_HANDLE	Error if `ctx` or `event` is an invalid handle.

calCtxFlush

Syntax	**CALresult calCtxFlush (CALcontext ctx)**	
Description	Flushes all the queues on the supplied context ctx. Calling calCtxFlush causes all queued commands to be submitted to the device.	
Results	CAL_RESULT_OK	Success.
	CAL_RESULT_ERROR	Error.

B.3.7 Error Reporting

Error reporting is encoded in the return code of nearly every platform function call. The CAL API can provide contextual information about an error.

calGetErrorString

Syntax	**const CALchar* calGetErrorString(void);**
Description	Returns a contextual string regarding the nature of the an error returned by a CAL API call. The error string represents global state to the CAL runtime. The error state is updated on every call to the CAL API. The error string is returned by the function call and is null terminated.

B.4 Extensions

The CAL API supports extensions to the core. Extensions are optional, and a CAL client can query their support. The extension mechanism provides future functionality and improvement without changing the overall ABI of the CAL libraries. Likewise, not all extensions are available on all platforms.

B.4.1 Extension Functions

The following is a description of the extension functions.

calExtSupported

Syntax	**CALresult calExtSupported (CALextid extid)**	
Description	Queries if an extension is supported by the implementation. The list of extensions is listed in Structures, on page B-25.	
Results	CAL_RESULT_OK	Extension is supported.
	CAL_RESULT_NOT_SUPPORTED	Extension is not supported.

calExtGetVersion

Syntax	**`CALresult calExtGetVersion(`** **`CALuint* major,`** **`CALuint* minor,`** **`CALextid extid)`**	
Description	Returns the version number of a supported extension. The format of the version number is in `major.minor` form. The list of extensions is listed in Section B.5.2, "Structures."	
Results	CAL_RESULT_OK	Success, and `major` and `minor` contain the returned values.
	CAL_RESULT_NOT_SUPPORTED	Extension is not supported.

calExtGetProc

Syntax	**`CALresult calExtGetProc(`** **`CALextproc* proc,`** **`CALextid extid,`** **`const CALchar* procname)`**	
Description	Returns a pointer to the function for the specified extension. The extension to the query is specified by the `extid` parameter. The name of the function to get a pointer to is specified by `procname`. The list of extensions is listed in Structures, on page B-25. The list of functions is in Section B.5, "CAL API Types," page 25.	
Results	CAL_RESULT_OK	Success, and `proc` contains a pointer to the function.
	CAL_RESULT_NOT_SUPPORTED	Error if either the `extid` is not valid or the function name was not found. On error, `proc` is null.

B.4.2 Interoperability Extensions

B.4.2.1 Direct3D 9 API

The following function calls are part of the Direct3D 9 API extension.

calD3D9Associate

Syntax	`CALresult calD3D9Associate(CALdevice dev, IDirect3DDevice9* d3dDevice)`	
Description	Initializes the CAL to Direct3D 9 interoperability, associating the CALdevice *dev* with the IDirect3DDevice9 *d3dDevice*.This function must be called before any other Direct3D 9 interoperability calls are made.	
Results	CAL_RESULT_ERROR	Interoperability not possible.
	CAL_RESULT_OK	Success.

calD3D9MapSurface

Syntax	`CALresult calD3D9MapSurface(CALresource* res, CALdevice dev, IDirect3DSurface9* surf, HANDLE shareHandle)`
Description	Maps the memory associated with IDirect3DSurface9 *surf* into the returned CALresource *res*. This function call can be used to map surfaces that are part of textures, render targets, or off-screen surfaces. The surface must have been created in the `D3DPOOL_DEFAULT` pool. Use only non-mipmapped textures with calD3D9MapSurface. The CAL resource format matches the D3DFORMAT. Once a resource has been created with `calD3D9MapSurface`, it can be used like any other CALresource. Before releasing the IDirect3DSurface9, the resource must be freed with `calResFree`. When running under Windows Vista, *shareHandle* must be the pSharedHandle returned when the surface or texture was created. When running under Windows XP, *shareHandle* must be 0.

Results		
	`CAL_RESULT_OK`	Success.
	`CAL_RESULT_ERROR`	Indicates that *surf* cannot be mapped on *dev*.

B.4.2.2 Direct3D 10 API

The following function calls are part of the Direct3D 10 API extension.

calD3D10Associate

Syntax	`CALresult calD3D10Associate(CALDevice dev, ID3D10Device* d3dDevice)`
Description	Initializes the CAL Direct3D 10 interoperability, associating the CALdevice *dev* with the ID3D10Device *d3dDevice*. This function must be called before any other Direct3D 10 interoperability calls are made.

Results		
	`CAL_RESULT_ERROR`	Interoperability not possible.
	`CAL_RESULT_OK`	Success.

calD3D10MapResource

Syntax	`CALresult calD3D10MapResource(CALresource* res, CALdevice dev, ID3D10Resource* d3dres, HANDLE shareHandle)`
Description	Maps the memory associated with *d3dres* into a CALresource, returned in *res*. The resource must have been created with the `D3D10_RESOURCE_MISC_SHARED` flag. Once a resource has been created with `calD3D10MapResource`, it can be used like any other CALresource. Before releasing the `ID3D10Resource`, the resource must be freed with `calResFree`. *shareHandle* must be obtained by getting an IDXGI resource interface from the D3D resource. The sharehandle then can be retrieved with `IDXGI::GetSharedHandle`.
Results	`CAL_RESULT_OK` — Success. `CAL_RESULT_ERROR` — Indicates that *surf* cannot be mapped on *dev*.

B.4.3 Counters

The following are descriptions of the counter functions.

calCtxCreateCounter

Syntax

```
CALresult calCtxCreateCounter(
    CALcounter* counter,
    CALcontext ctx,
    CALcountertype type)
```

Description

Create a counter object. The counter is created on the specified context `ctx` and is of type `type`. Supported counters are:

CAL_COUNTER_IDLE	Percentage of time the stream processor is idle between Begin/End delimiters.
CAL_COUNTER_INPUT_CACHE_HIT_RATE	Percentage of input memory requests that hit the cache.

Counter activity is bracketed by a Begin/End pair. All activity to be considered must be between `calCtxBeginCounter` and `calCtxEndCounter`. Any number of `calCtxRunProgram` calls can exist between the `Begin` and `End` calls.

Results

CAL_RESULT_OK	Success, and a handle to the counter is returned in `counter`.
CAL_RESULT_BAD_HANDLE	Error if `ctx` is an invalid handle.

calCtxDestroyCounter

Syntax	**CALresult calCtxDestroyCounter(CALcontext ctx, CALcounter counter)**
Description	Destroys a created counter object. The counter to destroy is specified by counter on the context specified by ctx. If a counter is destroyed between calCtxBeginCounter and calCtxEndCounter, CAL_RESULT_BUSY is returned.

Results	CAL_RESULT_OK	Success.
	CAL_RESULT_BAD_HANDLE	Error if called between Begin and End.

calCtxBeginCounter

Syntax	**CALresult calCtxBeginCounter(CALcontext ctx, CALcounter counter)**
Description	Initiates counting on the specified counter. Counters can be started only in a context. The counter is specified by counter. The context to start the counter on is specified by ctx.

Results	CAL_RESULT_OK	Success.
	CAL_RESULT_BAD_HANDLE	Error if either ctx or counter is an invalid handle.
	CAL_RESULT_ALREADY	Error if calCtxBeginCounter has been called on the same counter without ever calling calCtxEndCounter.
		On error, the state is as if calCtxBeginCounter had not been called.

calCtxEndCounter

Syntax

```
CALresult calCtxEndCounter(
    CALcontext ctx,
    CALcounter)
```

Description Ends counting on the specified counter. A counter can be ended only in the same context in which it was started. Counters can be ended once they are started by calCtxBeginCounter.

Results

CAL_RESULT_OK	Success.
CAL_RESULT_BAD_HANDLE	Error if either ctx or counter is an invalid handle.
CAL_RESULT_ERROR	Error if calCtxEndCounter is called without having called calCtxBeginCounter.
	On error, the CAL API behaves as if calCtxEndCounter had not been called.

calCtxGetCounter

Syntax

```
CALresult calCtxGetCounter(
    CALfloat* result,
    CALcontext ctx,
    CALcounter counter)
```

Description Retrieves the results of a counter. The value of the results is a floating point number between 0.0 and 1.0 whose meaning is shown in the description for calCtxCreateCounter, on page B-22. The results of a counter might not be available immediately. The counter results can be polled for availability, or the last calCtxRunProgram returned event can be polled for availability.

Results

CAL_RESULT_OK	Success, and result contains the result of the counter.
CAL_RESULT_BAD_HANDLE	Error if either ctx or counter is an invalid handle.
CAL_RESULT_PENDING	Counter results are not available.
	On error, result is 0.0.

B.5 CAL API Types

The following subsections detail the enums and structs for the CAL API.

B.5.1 Enums

CALcountertype

```
enum CALcountertype
{
    CAL_COUNTER_IDLE,
    CAL_COUNTER_INPUT_CACHE_HIT_RATE
};
```

CALextid

```
enum CALextid
{
    CAL_EXT_D3D9     = 0x1001, /* CAL/D3D9 interaction extension */
    CAL_EXT_OPENGL   = 0x1002, /* CAL/OpenGL interaction extension */
    CAL_EXT_D3D10    = 0x1003, /* CAL/D3D10 interaction extension */
    CAL_EXT_COUNTERS = 0x1004, /* CAL counter extension */
};
```

B.5.2 Structures

CALdeviceattribs

```
struct CALdeviceattribs
{
    CALuint struct_size;          /* client filled out size of CALdeviceattribs struct */
    CALtarget target;             /* asic identifier */
    CALuint physicalRAM;          /* amount of local GPU RAM in megabytes */
    CALuint uncachedRemoteRAM;    /* amount of uncached remote GPU memory in megabytes */
    CALuint cachedRemoteRAM;      /* amount of cached remote GPU memory in megabytes */
    CALuint engineClock;          /* GPU device clock rate in megahertz */
    CALuint memoryClock;          /* GPU memory clock rate in megahertz */
};
```

CALdevicestatus

```
struct CALdevicestatus
{
    CALuint struct_size;                /* client filled out size of struct */
    CALuint availLocalRAM;              /* available local RAM in megabytes */
    CALuint availUncachedRemoteRAM;     /* available uncached remote memory in megabytes */
    CALuint availcachedRemoteRAM;       /* available cached remote memory in megabytes */
};
```

B.6 Function Calls in Alphabetic Order

Table B.1 lists all function calls in alphabetic order, including the group to which each one belongs and the page that contains its complete description.

Table B.1 Function Calls in Alphabetic Order

Function	Group	Described on Page
calCtxBeginCounter	Counters	B-23
calCtxCreate	Context Management	B-13
calCtxCreateCounter	Counters	B-22
calCtxDestroy	Context Management	B-13
calCtxDestroyCounter	Counters	B-23
calCtxEndCounter	Counters	B-24
calCtxFlush	Computation	B-19
calCtxGetCounter	Counters	B-24
calCtxGetMem	Context Management	B-14
calCtxIsEventDone	Computation	B-18
calCtxReleaseMem	Context Management	B-14
calCtxRunProgram	Computation	B-17
calCtxSetMem	Context Management	B-14
calDeviceClose	Device Management	B-7
calDeviceGetAttribs	Device Management	B-6
calDeviceGetCount	Device Management	B-5
calDeviceGetStatus	Device Management	B-7
calDeviceOpen	Device Management	B-6
calExtGetProc	Core Functions	B-20
calExtGetVersion	Core Functions	B-20
calExtSupported	Core Functions	B-19
calGetErrorString	Error Reporting	B-19
calGetVersion	System Component	B-5
calImageRead	Loader	B-17
calInit	System Component	B-4
calMemCopy	Computation	B-18
calModuleGetEntry	Loader	B-16
calModuleGetName	Loader	B-16
calModuleLoad	Loader	B-15
calModuleUnload	Loader	B-15
calResAllocLocal1D	Memory Management	B-10
calResAllocLocal2D	Memory Management	B-8
calResAllocRemote1D	Memory Management	B-11
calResAllocRemote2D	Memory Management	B-9
calResFree	Memory Management	B-11
calResMap	Memory Management	B-12
calResUnmap	Memory Management	B-12
calShutdown	System Component	B-5

Appendix C
Supported Devices

The devices supported by the current version of the Stream Computing software are:

- ATI Radeon HD 3870
- AMD FireStream 9170
- ATI Radeon HD 4850
- AMD FireStream 9250
- ATI FireGL V7700
- ATI Radeon HD 4870

Appendix D Introduction to 3D Graphics and Shader Terminology

The following descriptions provide an introductory explanation of some concepts and terminology used in stream computing. These descriptions try, by simplification, to make stream computing terminology understandable to CPU programmers. Stream computing is derived from 3D graphics programming; thus, some understanding of GPU programming is useful.

D.1 Shaders

Shader programs are what define the programmers view of a GPU. The notion of a shader or a shader program originated from the concept of adding realistic lighting to a 3D object as a final step before displaying the image on the screen. Imagine an array of pixels in an X-Y grid. A program loop iterates over each X-Y location, reading each pixel, modifying it based on some algorithmic light source, and then writing the modified pixel to the final frame buffer that is used to refresh the screen.

Defining the problem in this way allows for some extreme optimizations if simple rules are followed.

1. The input buffer can only be read from, not written to.
2. During the shading step, the output buffer can only be written to, not read from. (It can be read later as the image is being displayed.)
3. Each loop iteration only generates a single pixel as its output.

These rules eliminate dependencies between successive iterations of the loop. This allows specialized hardware to eliminate the loop setup and iteration mechanisms, as well as execute every iteration of the loop simultaneously (or with the available parallel hardware).

D.2 Domain of Execution

During the processing of a shader the *domain of execution* is merely the specification of the X-Y output grid being computed. The domain of execution could be a entire frame or some portion of it at the programmer's discretion.

D.3 Geometry and Vertices

Traditional 3D processing starts with the geometry processing step. A a collection of vertices (x,y,z coordinates) in sequence define small (relatively) triangles in the

3D space. Each 3D object is a mesh of such triangles (or polygons). Triangles are used because 3 points in space are the minimum required to define a surface. For example, an object like a flat rectangular table top can be made up of 2 adjacent triangles, 4 vertices in total that share two vertices.

A vertex shader program can alter the coordinates and/or properties of a vertex using approximately the same rules that allows for parallelism in a pixel shader program. Each vertex requires the 3 coordinate values X, Y and Z to define its position in space (plus a 4^{th} “W” component which is normally set to 1.0). These four floating point values are stored in a 4-wide structure which can be defined as a vec4. Transformations of these vec4 arrays are done using 4x4 matrices. These details are only important because of the fact that the GPU hardware is highly optimized at performing these types of operations on this size of data. Arranging your data in a similar way is not required but can give you a large performance advantage.

Additionally, per pixel data is stored in a vec4 format as RGBA as red, green, blue and alpha components, where the alpha represents a transparency from 0.0 to 1.0. Various low-level GPU instructions may refer to data in registers or variables using a nomenclature, such as var.xyzw or var.rgba. In both cases the variable var is assumed to be of the type vec4 with the first 32-bit floating point value indicated interchangeably by x or r, the second element y or g, etc.

Glossary of Terms

Term	**Description**
*	Any number of alphanumeric characters in the name of a microcode format, microcode parameter, or instruction.
< >	Angle brackets denote streams.
[1,2)	A range that includes the left-most value (in this case, 1) but excludes the right-most value (in this case, 2).
[1,2]	A range that includes both the left-most and right-most values (in this case, 1 and 2).
{x | y}	One of the multiple options listed. In this case, x or y.
0.0	A single-precision (32-bit) floating-point value.
1011b	A binary value, in this example a 4-bit value.
7:4	A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first.
ABI	Application Binary Interface.
ACML	AMD Core Math Library. Includes implementations of the full BLAS and LAPACK routines, FFT, Math transcendental and Random Number Generator routines, stream processing backend for load balancing of computations between the CPU and stream processor.
AL	Loop register. A 3-element vector (x, y and z) used to count iterations of a loop.
ALU	Arithmetic Logic Unit. Responsible for arithmetic operations like addition, subtraction, multiplication, division, and bit manipulation on integer and floating point values. In stream computing, these are known as *stream cores*.
AMD Stream™ SDK	A complete software development suite from AMD for developing applications for AMD Stream Processors. Currently, AMD Stream SDK includes Brook+ and CAL.
AR	Address register.
aTid	Absolute thread id. It is the ordinal count of all threads being executed (in a draw call).
b	A bit, as in *1Mb* for one megabit, or *lsb* for least-significant bit.
B	A byte, as in *1MB* for one megabyte, or *LSB* for least-significant byte.
BLAS	Basic Linear Algebra Subroutines.
branch granularity	The number of threads executed during a branch. For AMD, branch granularity is equal to wavefront granularity.
brcc	Source-to-source meta-compiler that translates Brook programs (.br files) into device-dependent kernels embedded in valid C++ source code that includes CPU code and stream processor device code, which later are linked into the executable.

Term	Description
Brook+	A high-level language derived from C which allows developers to write their applications at an abstract level without having to worry about the exact details of the hardware. This enables the developer to focus on the algorithm and not the individual instructions run on the stream processor. Brook+ is an enhancement of Brook, which is an open source project out of Stanford. Brook+ adds additional features available on AMD Stream Processors and provides a CAL backend.
brt	The Brook runtime library that executes pre-compiled kernel routines invoked from the CPU code in the application.
burst mode	The limited write combining ability. See write combining.
byte	Eight bits.
cache	A read-only or write-only on-chip or off-chip storage space.
CAL	Compute Abstraction Layer. A device-driver library that provides a forward-compatible interface to AMD stream processor devices. This lower-level API gives users direct control over the hardware: they can directly open devices, allocate memory resources, transfer data and initiate kernel execution. CAL also provides a JIT compiler for AMD IL.
channel	An element in a vector.
clause	A group of instructions that are of the same type (all stream core, all fetch, etc.) executed as a group. A clause is part of a CAL program written using the stream processor ISA. Executed without pre-emption.
clause size	The total number of slots required for an stream core clause.
clause temporaries	Temporary values stored at GPR that do not need to be preserved past the end of a clause.
clear	To write a bit-value of 0. Compare “set”.
command	A value written by the host processor directly to the stream processor. The commands contain information that is not typically part of an application program, such as setting configuration registers, specifying the data domain on which to operate, and initiating the start of data processing.
command processor	A logic block in the R600 that receives host commands (see Figure 1.4), interprets them, and performs the operations they indicate.
component	An element in a vector.
compute shader	Similar to a pixel shader, but exposes data sharing and synchronization.
constant buffer	Off-chip memory that contains constants. A constant buffer can hold up to 1024 4-element vectors. There are fifteen constant buffers, referenced as cb0 to cb14. An immediate constant buffer is similar to a constant buffer. However, an immediate constant buffer is defined within a kernel using special instructions. There are fifteen immediate constant buffers, referenced as icb0 to icb14.
constant cache	A constant cache is a hardware object (off-chip memory) used to hold data that remains unchanged for the duration of a kernel (constants). “Constant cache” is a general term used to describe constant registers, constant buffers or immediate constant buffers.
constant registers	On-chip registers that contain constants. The registers are organized as four 32-bit elements of a vector. There are 256 such registers, each one 128-bits wide.
context	A representation of the state of a CAL device.
core clock	See engine clock. The clock at which the stream processor stream core runs.

Term	Description
CPU	Central Processing Unit. Also called host. Responsible for executing the operating system and the main part of the application. The CPU provides data and instructions to the stream processor.
CRs	Constant registers. There are 512 CRs, each one 128 bits wide, organized as four 32-bit values.
CS	Compute shader. A new shader type for R7xx, analogous to VS/PS/GS/ES
CTM	Close-to-Metal. A thin, HW/SW interface layer. This was the predecessor of the AMD CAL.
DC	Data Copy Shader.
device	A *device* is an entire AMD stream processor.
DMA	Direct-memory access. Also called DMA engine. Responsible for independently transferring data to, and from, the stream processor's local memory. This allows other computations to occur in parallel, increasing overall system performance.
domain of execution	A specified rectangular region of the output buffer to which threads are mapped.
DPP	Data-Parallel Processor.
element	(1) A 32-bit piece of data in a "vector". (2) A 32-bit piece of data in an array. (3) One of four data items in a 4-component register.
engine clock	The clock driving the stream core and memory fetch units on the stream processor stream processor core.
enum(7)	A seven-bit field that specifies an enumerated set of decimal values (in this case, a set of up to 27 values). The valid values can begin at a value greater than, or equal to, zero; and the number of valid values can be less than, or equal to, the maximum supported by the field.
event	A token sent through a pipeline that can be used to enforce synchronization, flush caches, and report status back to the host application.
export	To write data from GPRs to an output buffer (scratch, ring, stream, frame or global buffer, or to a register), or to read data from an input buffer (a "scratch buffer" or "ring buffer") to GPRs. The term "export" is a partial misnomer because it performs both input and output functions. Prior to exporting, an allocation operation must be performed to reserve space in the associated buffer.
FFT	Fast Fourier Transform.
flag	A bit that is modified by a CF or stream core operation and that can affect subsequent operations.
FLOP	Floating Point Operation.
frame	A single two-dimensional screenful of data, or the storage space required for it.
frame buffer	Off-chip memory that stores a frame.
FS	Fetch subroutine. A global program for fetching vertex data. It can be called by a "vertex shader" (VS), and it runs in the same thread context as the vertex program, and thus is treated for execution purposes as part of the vertex program. The FS provides driver independence between the process of fetching data required by a VS, and the VS itself. This includes having a semantic connection between the outputs of the fetch process and the inputs of the VS.

Term	Description
function	A subprogram called by the main program or another function within an AMD IL stream. Functions are delineated by `FUNC` and `ENDFUNC`.
gather	Reading from arbitrary memory locations by a thread.
gather stream	Input streams are treated as a memory array, and data elements are addressed directly.
global buffer	Memory space containing the arbitrary address locations to which uncached kernel outputs are written. Can be read either cached or uncached. When read in uncached mode, it is known as mem-import. Allows applications the flexibility to read from and write to arbitrary locations in input buffers and output buffers, respectively.
GPGPU	General-purpose stream processor. A stream processor that performs general-purpose calculations.
GPR	General-purpose register. GPRs hold vectors of either four 32-bit IEEE floating-point, or four 8-, 16-, or 32-bit signed or unsigned integer or two 64-bit IEEE double precision data elements (values). These registers can be indexed, and consist of an on-chip part and an off-chip part, called the “scratch buffer,” in memory.
GPU	Graphics Processing Unit. An integrated circuit that renders and displays graphical images on a monitor. Also called Graphics Hardware, Stream Processor, and Data Parallel Processor.
GPU engine clock frequency	Also called 3D engine speed.
GS	Geometry Shader.
GSA	GPU ShaderAnalyzer. A performance profiling tool for developing, debugging, and profiling stream kernels using high-level stream computing languages.
HAL	Hardware Abstraction Layer.
host	Also called CPU.
iff	If and only if.
IL	Intermediate Language. In this manual, the AMD version: AMD IL. A pseudo-assembly language that can be used to describe kernels for stream processors. AMD IL is designed for efficient generalization of stream processor instructions so that programs can run on a variety of platforms without having to be rewritten for each platform.
in flight	Stalled phase of a thread.
instruction	A computing function specified by the *code* field of an IL_OpCode token. Compare “opcode”, “operation”, and “instruction packet”.
instruction packet	A group of tokens starting with an IL_OpCode token that represent a single AMD IL instruction.
int(2)	A 2-bit field that specifies an integer value.
ISA	Instruction Set Architecture. The complete specification of the interface between computer programs and the underlying computer hardware.
kernel	A small, user-developed program that is run repeatedly on a stream of data. A parallel function that operates on every element of input streams. A device program is one type of kernel. Unless otherwise specified, an AMD stream processor program is a kernel composed of a main program and zero or more functions. Also called Shader Program.

Term	**Description**
LAPACK	Linear Algebra Package.
LERP	Linear Interpolation.
local memory fetch units	Dedicated hardware that a) processes fetch instructions, b) requests data from the memory controller, and c) loads registers with data returned from the cache. They are run at stream processor stream core or engine clock speeds. Formerly called texture units.
LOD	Level Of Detail.
loop index	A register initialized by software and incremented by hardware on each iteration of a loop.
lsb	Least-significant bit.
LSB	Least-significant byte.
MAD	Multiply-Add.
mask	(1) To prevent from being seen or acted upon. (2) A field of bits used for a control purpose.
MBZ	Must be zero.
mem-export	An AMD IL term random writes to the global buffer.
mem-import	Uncached reads from the global buffer.
memory clock	The clock driving the memory chips on the stream processor.
MIMD	Multiple Instruction Multiple Data. – Multiple SIMD units operating in parallel (Multi-Processor System) – Distributed or shared memory
MRT	Multiple Render Target. One of multiple areas of local stream processor memory, such as a “frame buffer”, to which a graphics pipeline writes data.
MSAA	Multi-Sample Anti-Aliasing.
msb	Most-significant bit.
MSB	Most-significant byte.
normalized	A numeric value in the range [a, b] that has been converted to a range of 0.0 to 1.0 using the formula: normalized value = value/ (b–a+ 1)
opcode	The numeric value of the *code* field of an “instruction”. For example, the opcode for the CMOV instruction is decimal 16 (10h).
opcode token	A 32-bit value that describes the operation of an instruction.
operation	The function performed by an “instruction”.
PaC	Parameter Cache.
PCI Express	A high-speed computer expansion card interface used by modern graphics cards, stream processors and other peripherals needing high data transfer rates. Unlike previous expansion interfaces, PCI Express is structured around point-to-point links. Also called PCIe.
PoC	Position Cache.

Term	Description
pre-emption	The act of temporarily interrupting a task being carried out on a computer system, without requiring its cooperation, with the intention of resuming the task at a later time.
primitive	A grouping of between one, two, or three vertices to form a geometric construct—a point, line segment, or triangle, respectively—that covers some number of fragments or pixels (points on an integer grid).
processor	Unless otherwise stated, the AMD Stream Processor and AMD Data Parallel Processor.
program	Unless otherwise specified, a program is a set of instructions that can run on the AMD Stream Processor/AMD Data Parallel Processor. A device program is a type of kernel.
PS	Pixel Shader.
quad	Group of 2x2 threads in the domain. Always processed together.
rasterization	The process of mapping threads from the domain of execution to the SIMD engine. This term is a carryover from graphics, where it refers to the process of turning geometry, such as triangles, into pixels.
rasterization order	The order of the thread mapping generated by rasterization.
RB	Ring Buffer.
register	A 32-bit address mapped piece of memory for accessing processor features.
relative	Referencing with a displacement (also called offset) from an index register or the loop index, rather than from the base address of a program (the first CF instruction).
render backend unit	The hardware units in a stream processor stream processor core responsible for writing the results of a kernel to output streams by writing the results to an output cache and transferring the cache data to memory.
resource	A block of memory used for input to, or output from, a kernel.
ring buffer	An on-chip buffer that indexes itself automatically in a circle.
Rsvd	Reserved.
sampler	A structure that contains information necessary to access data in a resource. Also called Fetch Unit.
SC	Scan Converter.
scalar	A single data element, unlike a vector which contains a set of two or more data elements.
scatter	Writes (by uncached memory) to arbitrary locations.
scatter write	Kernel outputs to arbitrary address locations. Must be uncached. Must be made to a memory space known as the global buffer.
scratch buffer	A variable-sized space in off-chip-memory that stores some of the “GPRs”.
set	To write a bit-value of 1. Compare “clear”.
shader processor	Also called thread processor.
shader program	User developed program. Also called kernel.
SIMD	Single instruction multiple data. – Each SIMD receives independent stream core instructions. – Each SIMD applies the instructions to multiple data elements.

Term	Description
SIMD Engine	A collection of thread processors, each of which executes the same instruction per cycle.
SIMD pipeline	A hardware block consisting of five stream cores, one stream core instruction decoder and issuer, one stream core constant fetcher, and support logic. All parts of a SIMD pipeline receive the same instruction and operate on different data elements.
Simultaneous Instruction Issue	Input, output, fetch, stream core, and control flow per SIMD engine.
SPU	Shader processing unit.
stage	A sampler and resource pair.
stream	A collection of data elements of the same type that can be operated on in parallel.
stream buffer	A variable-sized space in off-chip memory that stores an instruction stream. It is an output-only buffer, configured by the host processor. It does not store inputs from off-chip memory to the processor.
stream core	The fundamental, programmable computational units, responsible for performing integer, single, precision floating point, double precision floating point, and transcendental operations. They execute VLIW instructions for a particular thread. Each stream processor stream core handles a single instruction within the VLIW instruction.
stream operator	A node that can restructure data.
stream processor	A parallel processor capable of executing multiple threads of a kernel in order to process streams of data.
swizzling	To copy or move any element in a source vector to any element-position in a destination vector. Accessing elements in any combination.
thread	One invocation of a kernel corresponding to a single element in the domain of execution.
thread group	It contains one or more thread blocks. Threads in the same thread-group but different thread-blocks might communicate to each through global per-stream processor shared memory. This is a concept mainly for global data share (GDS) which is not discussed in this note.
thread processor	The hardware units in a SIMD engine responsible for executing the threads of a kernel. It executes the same instruction per cycle. Each thread processor contains multiple stream cores. Also called shader processor.
thread-block	A group of threads which might communicate to each other through local per SIMD shared memory. It can contain one or more wavefronts (the last wavefront can be a partial wavefront). A thread-block (i.e. all its wavefronts) can only run on one SIMD engine. However, multiple thread blocks can share a SIMD engine, if there are enough resources to fit them in.
Tid	Thread id within a thread block. An integer number from 0 to Num_threads_per_block-1
token	A 32-bit value that represents an independent part of a stream or instruction.
uncached read/write unit	The hardware units in a stream processor responsible for handling uncached read or write requests from local memory on the stream processor.
vector	(1) A set of up to four related values of the same data type, each of which is an element. For example, a vector with four elements is known as a “4-vector” and a vector with three elements is known as a “3-vector”. (2) See “AR”.

Term	Description
vertex	A unit of data.
VLIW design	Very Long Instruction Word. – Co-issued up to 6 operations (5 stream cores + 1 FC) – 1.25 Machine Scalar operation per clock for each of 64 data elements – Independent scalar source and destination addressing
VS	Vertex Shader.
wavefront	Group of threads executed together on a single SIMD engine. Composed of quads. A full wavefront contains 64 threads; a wavefront with fewer than 64 threads is called a partial wavefront.
write combining	Combining several smaller writes to memory into a single larger write to minimize any overhead associated with write commands.