Technical Overview

ATI Stream Computing

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

1 The ATI Stream Computing Programming Model

The ATI Stream Computing Model includes a software stack and the ATI Stream processors. Figure 1 illustrates the relationship of the ATI Stream Computing components.

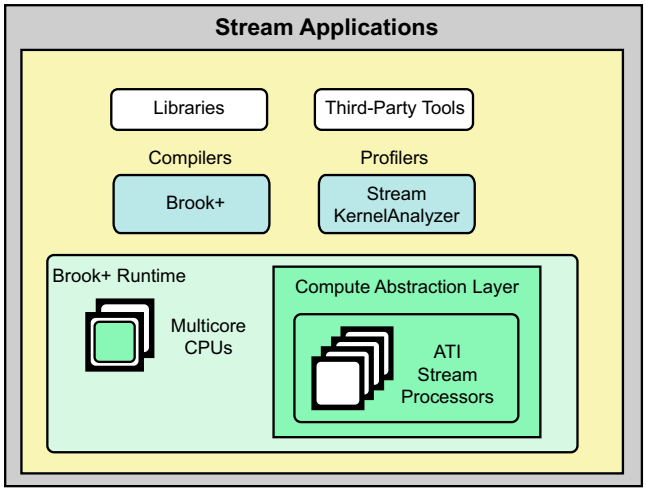

Figure 1 ATI Stream Software Ecosystem

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

The software includes the following components:

- Compilers – like the Brook+ compiler with extensions for ATI devices.
- Device Driver for stream processors – ATI Compute Abstraction Layer (CAL).[2]
- Performance Profiling Tools – Stream KernelAnalyzer.
- Performance Libraries – AMD Core Math Library (ACML) for optimized domain-specific algorithms.

1. A *stream* is a collection of data elements of the same type that can be operated on in parallel.
2. When using CAL, it might not be necessary to use Brook+; instead, it is possible to use ATI IL.

The latest generation of ATI 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 ATI Stream Computing programming model and the mapping of threads to thread processors is shown in Figure 2 (also see Figure 9).

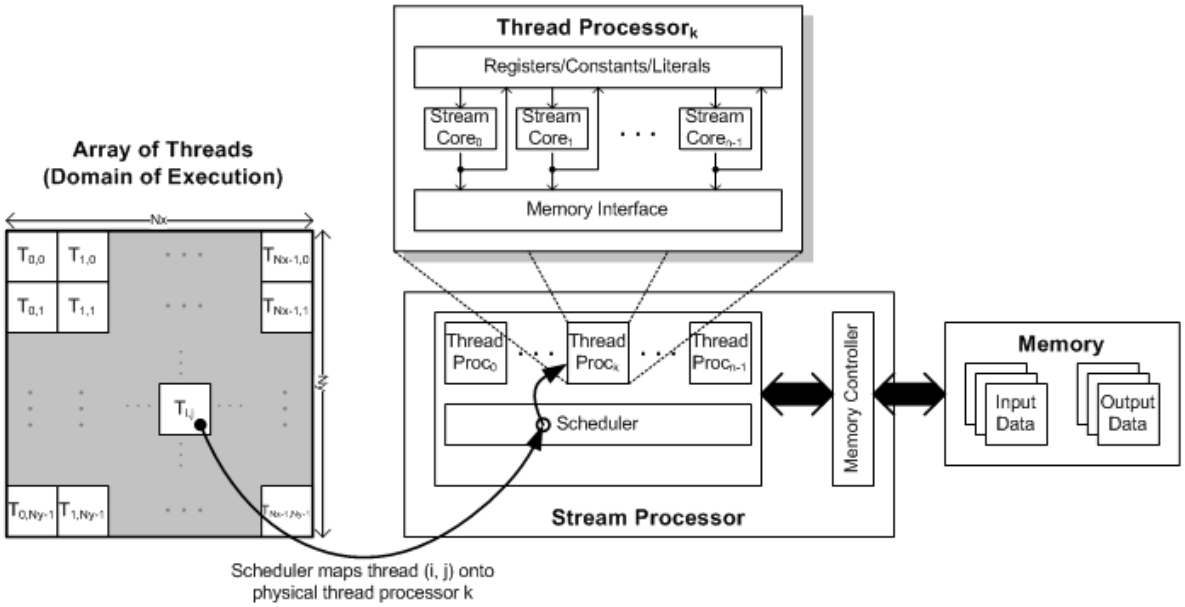

Figure 2 Simplified ATI Stream Computing Programming Model

1.1 Pseudo Code Explanation of ATI Stream Computing

Another way to explain the ATI 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 ATI 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 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 ATI 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.

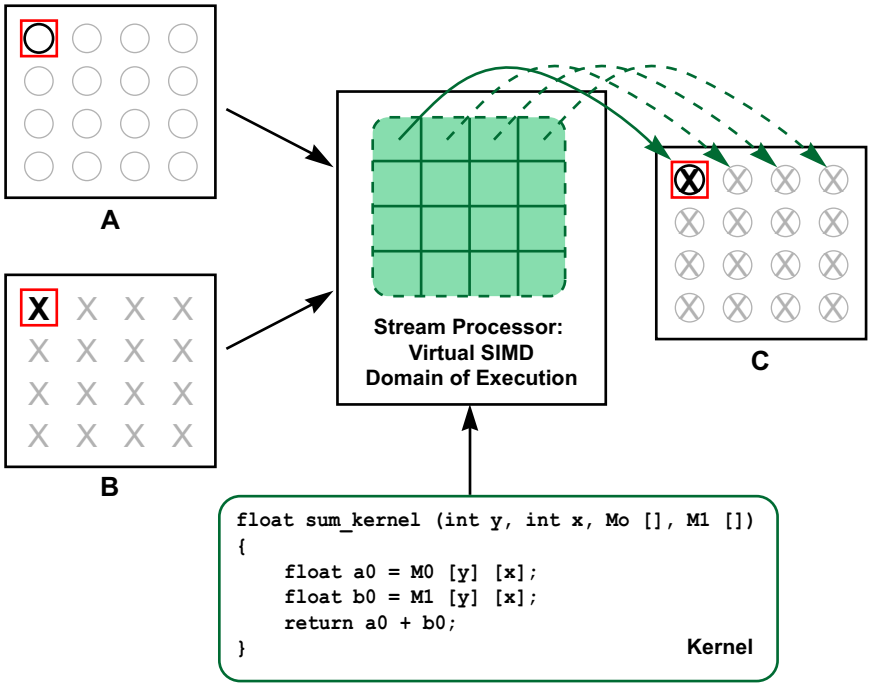

Figure 3 Stream Processor Execution

Matrix Multiply – This example multiplies two matrices (see Figure 4). This shows how some understanding of the hardware can improve performance.

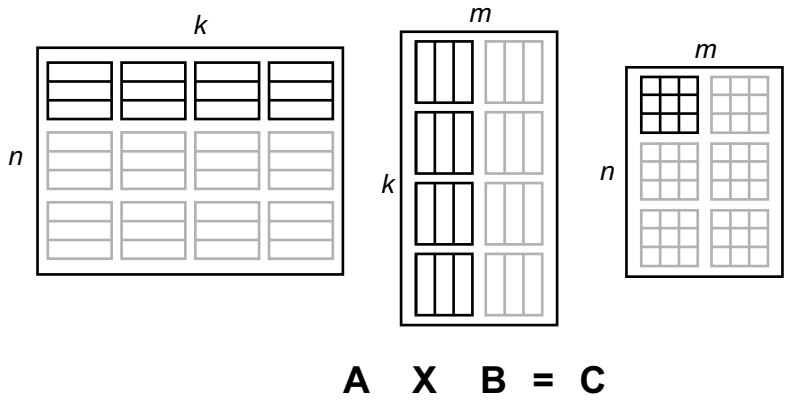

Figure 4 Matrix Multiply *(nxk) X (kxm)*

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 the ATI 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.

1.2 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 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.

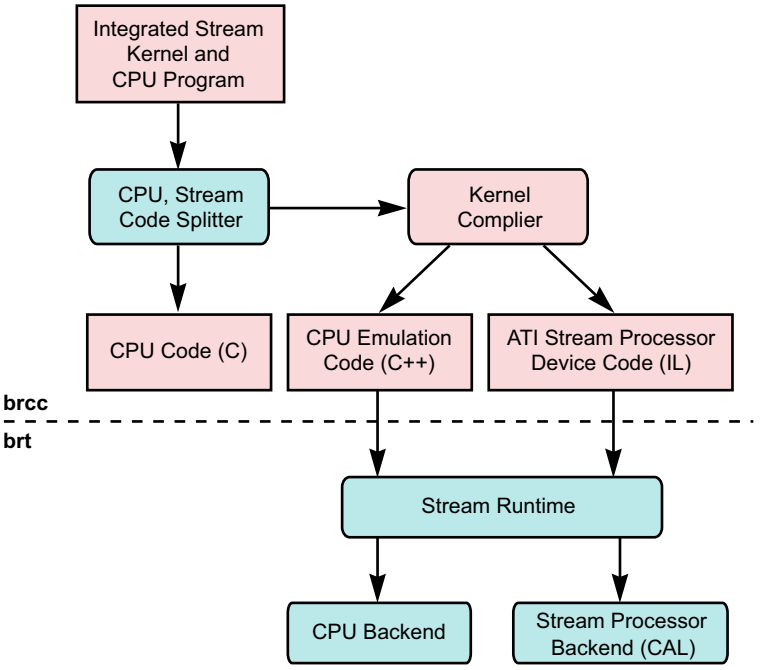

Figure 5 Brook+ Language Elements

ATI has enhanced brcc to produce the virtual instruction set architecture (ISA), called the ATI IL (for *I*ntermediate *L*anguage). ATI also has enhanced the brt with a backend optimized for ATI Stream processors using the CAL driver (see Section 1.3, “ATI Compute Abstraction Layer (CAL),” page 8).

1.3 ATI Compute Abstraction Layer (CAL)

The ATI Compute Abstraction Layer (CAL) is a device driver library that provides a forward-compatible interface to ATI Stream processors (see Figure 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

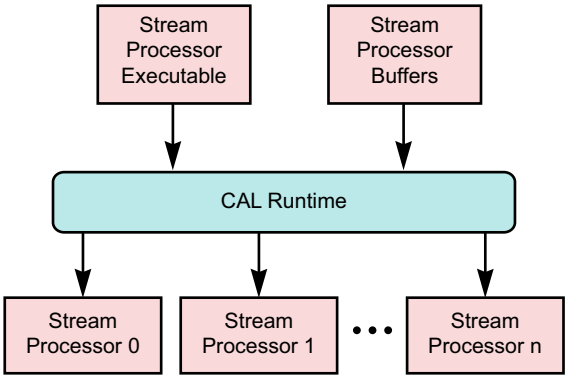

Figure 6 CAL Functionality

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 ATI IL and generates optimized code for the target architecture. It also provides access to device-specific features.

1.4 Stream KernelAnalyzer (SKA)

The Stream KernelAnalyzer (SKA) is a performance-profiling tool developers can use to develop and profile stream kernels. It can be downloaded for free from the ATI developer web pages, http://developer.amd.com/gpu/ska/Pages/default.aspx.

Features of the SKA include:

- Quick syntax checking of programs written in Brook+.
- Online kernel compilation to generate the equivalent ATI 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 Stream KernelAnalyzer has a simple graphical user interface. Figure 7 shows an example kernel, that was written in Brook+ and is converted to ATI IL. The generated ATI IL can be sent to the CAL runtime compiler for object code generation and subsequent execution.

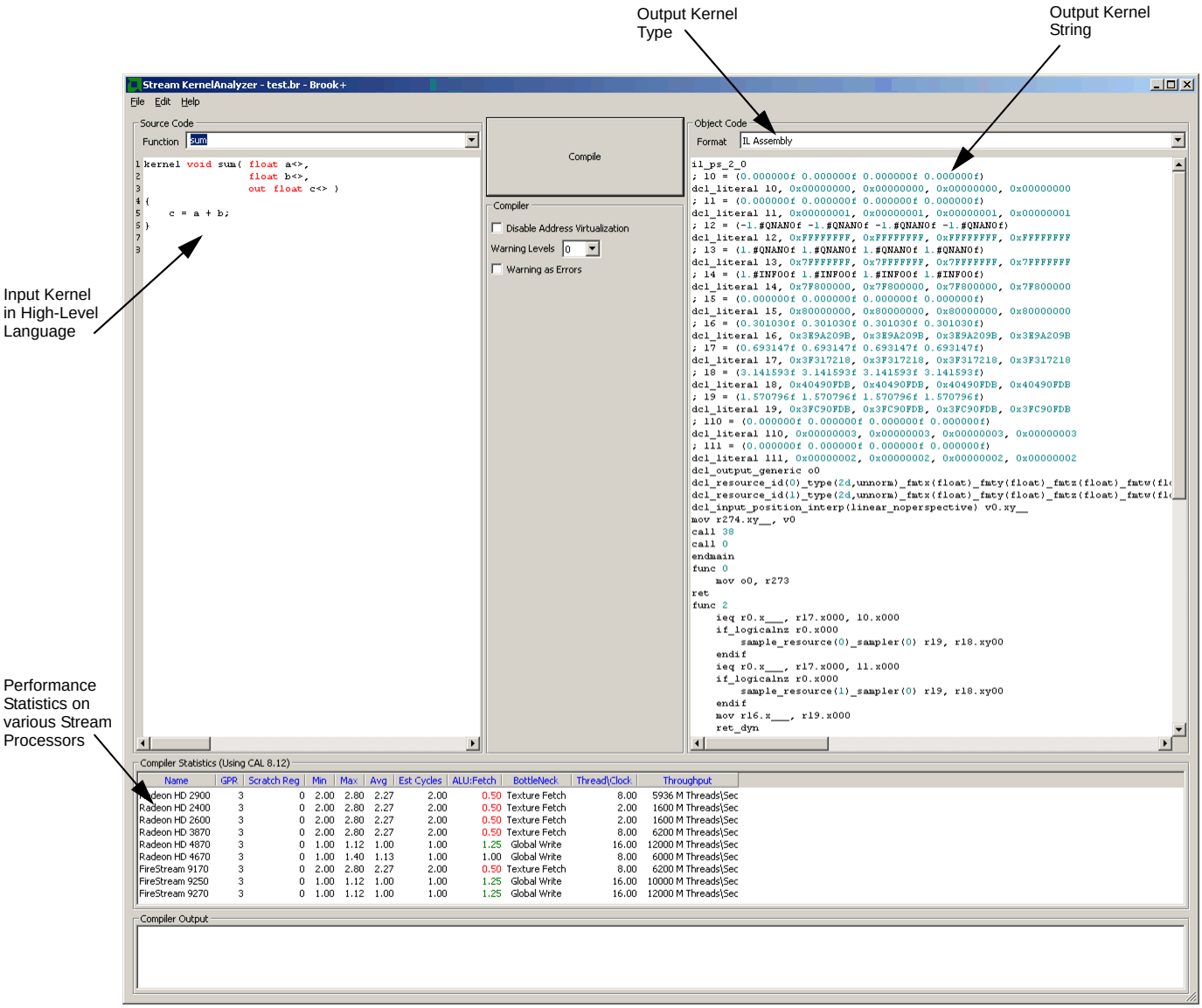

Figure 7 SKA User Interface Example

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.

1.5 AMD Core Math Library (ACML)

The ACML includes a collection of commonly used mathematical software routines. It is optimized for ATI 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.[3] This is done at runtime.

1.6 Compute Kernels

Multiple kernel types are executable on ATI Stream processors, include *vertex*, *pixel*, and *geometry*. Pixel kernels sometimes were used for non-graphics computing. Now, hardware that takes advantage of the processing power of kernels has been developed specifically for computational tasks. This hardware executes *compute kernels,* a specific kernel type that does not fit in the traditional graphics pipeline. The compute kernel can be used for graphics, but its strength lies in using it for non-graphics fields such as physics, AI, modeling, HPC, and various other computationally intensive applications.

Compute kernels differ from pixel kernels in the following areas.

- Indexing

  In a compute kernel, the indexing method is switched to a linear index between one and three dimensions, as specified by the user. This gives the programmer more flexibility when writing kernels. On the current generation of ATI Stream processors, only one dimension is natively supported; the other two dimensions are handled by address translation.

- Wavefronts and Groups

  Wavefronts and *groups* are two concepts relating to compute kernels that provide data parallel granularity. Wavefronts are hardware threads that execute N number of threads in parallel, where N is specific to the hardware chip (for example, on the ATI Radeon HD4870 it is 64). A wavefront processes a single instruction over all of the threads at the same time.

  Grouping is a higher-level granularity of data parallelism that is enforced in software, not hardware. A group is a set number of threads that execute blocks of code together in parallel before another group can execute the same block of code.

3. 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.

- Memory Access Pattern

  The access pattern for pixel kernels is in a hierarchical Z and is tuned for tiled memory performance. The access pattern for a compute kernel is linear across each row before moving to next row. This affects performance, since pixel kernels have implicit blocking, and compute kernels do not.

- Thread Spawn Rate

  In a compute shader, the thread spawn rate is linear. This means that on a chip with N threads per wavefront, the first N threads go to wavefront 1, the second N threads go to wavefront 2, etc.

- Local Data Store (LDS)

  The LDS is a write-private, read-public model: a thread can write only to its own memory space but can read from the memory space of any thread in the same group.

- Shared Registers

  Shared registers are a method of sharing data at a lower level than the LDS. The LDS shares data at the group level, but shared registers share data at the wavefront level. The shared registers are unique to the index of a wavefront and share data between wavefronts; this enables vertical sharing between all the wavefronts that run on a SIMD. This feature allows sharing between groups; however, one constraint is that shared registers only guarantee atomicity during the same instruction.

2 Stream Processor Hardware Functionality

Figure 8 shows a simplified block diagram of a generalized stream processor.

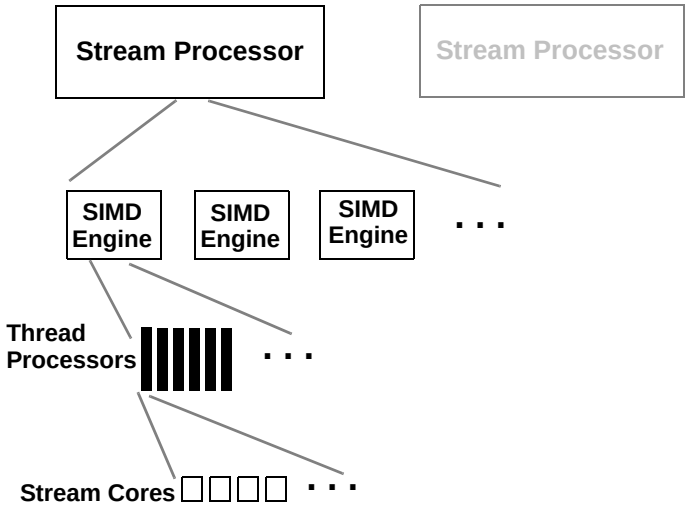

Figure 8 Generalized Stream Processor Structure

2.1 The Stream Processor

Figure 9 is a simplified diagram of an ATI 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 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.

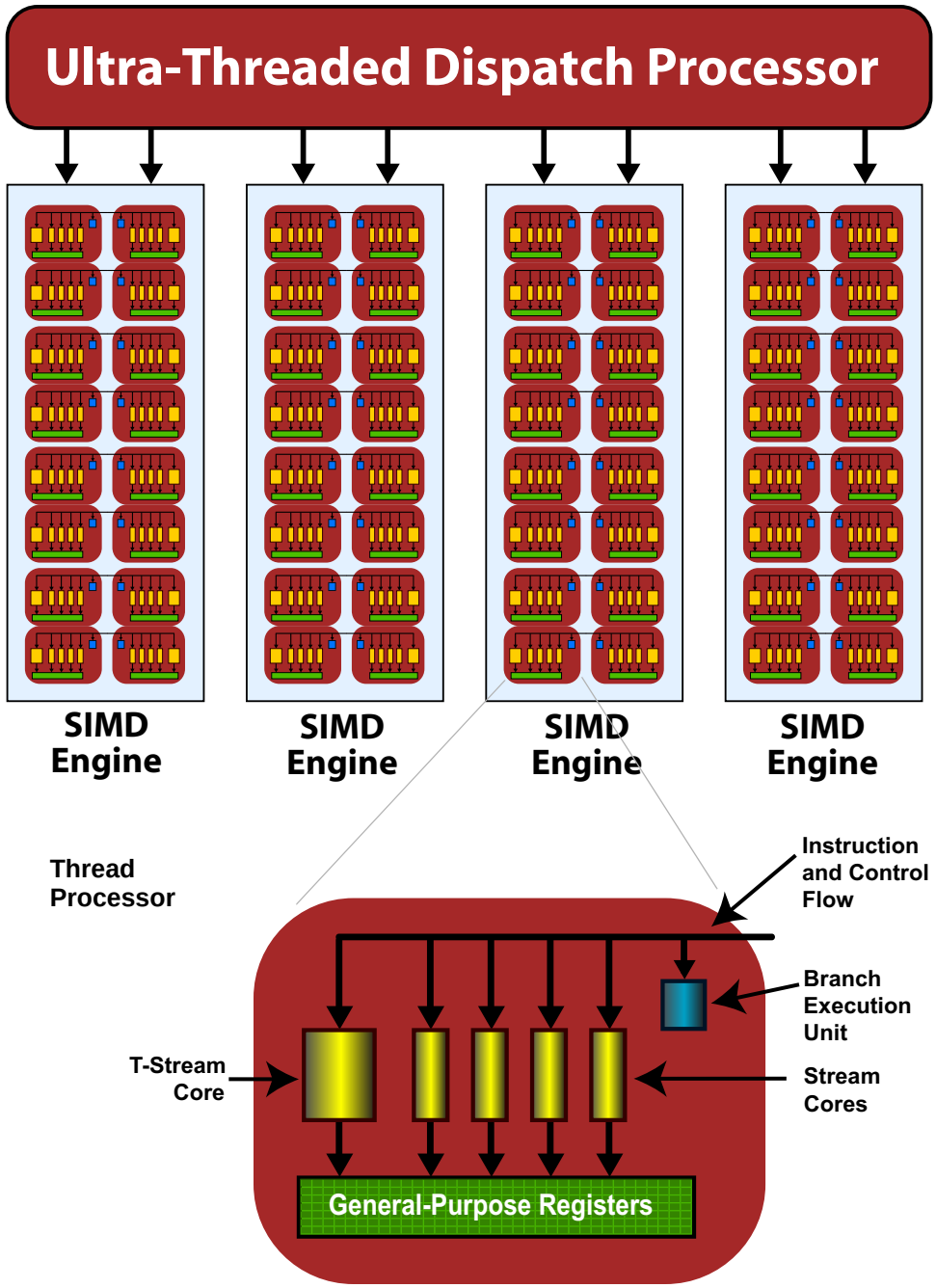

Figure 9 Simplified Block Diagram of the Stream Processor[4]

A thread processor is arranged as a five-way VLIW processor (see bottom of Figure 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.)[5]. Double-precision floating point operations are processed by connecting four of the stream cores (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.

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.

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 ATI 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*.

4. As described later, much of this is transparent to the programmer.
5. For the actual operations, see the ATI *Compute Abstraction Layer (CAL) Technology Intermediate Language (IL) Reference Manual*.

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. 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*.

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 10). *Rasterization* is the process of mapping threads from the domain of execution to SIMD engines[6].

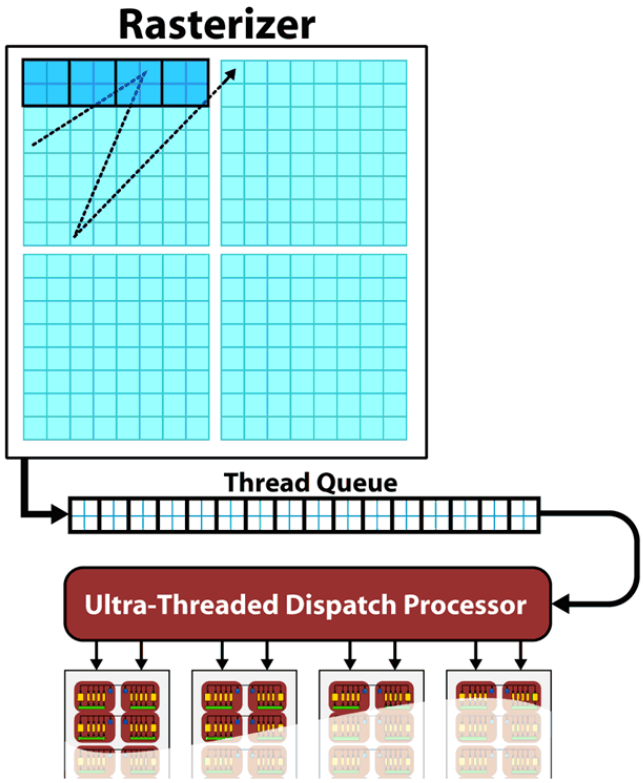

Figure 10 Rasterization of Threads to SIMD Engines

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

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.

2.4.2 Thread Optimization

ATI 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 2.5.1, “Memory Access,” page 16.

2.5 Memory Architecture and Access

There are three memory domains for developing stream processor applications: host (CPU) memory, PCIe memory, local (stream processor) memory.

Host (CPU) memory is used by applications. It is only available to the user’s applications; the GPU cannot access it. This is where the application’s data structures and program data reside.

PCIe memory is a section of host (CPU) memory set aside for PCIe use. It is accessible from the host program and the stream process and can be modified by both. Modifying this memory requires synchronization between the stream processor and CPU, usually with the `calCtxIsEventDone` API call. Brook+ makes this transparent.

Local (stream processor) memory is the GPU version of host memory. It is only accessible by the stream processor and cannot be accessed through the CPU.

There are three ways to copy data to stream processor memory:

- Implicitly through `calResMap`/`calResUnmap`.
- Explicitly through `calCtxMemCopy`.
- Explicitly with a custom kernel that reads from PCIe memory and writes to stream processor memory.

The important consideration when using these interfaces is the amount of copying involved. In a program that does not handle memory transfers (such as all of the samples), there is a two copy processes: between host and PCIe, and between PCIe and stream processor. This is why there is a large performance difference between the system GFLOPS and the kernel GFLOPS.

With proper memory transfer management and the use of system pinned memory (host/CPU memory remapped to the PCIe memory space) through `calCtxResCreate` in the `cal_ext.h`, copying between host (CPU) memory and PCIe memory can be skipped. Note that this is not an

easy API call to use and comes with many constraints, such as page boundary and memory alignment.

Double copying lowers the overall system memory bandwidth. Copies between host (CPU) memory and PCIe memory usually are in the hundreds of MBps; those between the PCIe memory and stream processor memory are in the GBps range. On-chip memory bandwidth is in the tens to hundred GBps range. In stream processor programming, pipeline executions and copies, or other techniques, to reduce these copy bottlenecks.

CAL resources used by Brook+ can be located in two of the three memory locations (PCIe memory, local stream processor memory).

To create a local (stream processor) memory space, use `calResAllocLocal` API function; to create a PCIe memory space, use the `calResAllocRemote` API function.

2.5.1 Memory Access

Accessing stream processor local memory typically is an order of magnitude faster than accessing remote (system or CPU) memory. However, stream cores (see Figure 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.

2.5.2 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.

2.5.3 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.

2.5.4 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.

2.5.5 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.

2.5.6 Memory Tiling

There are many possible physical memory layouts for data streams. ATI 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 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.

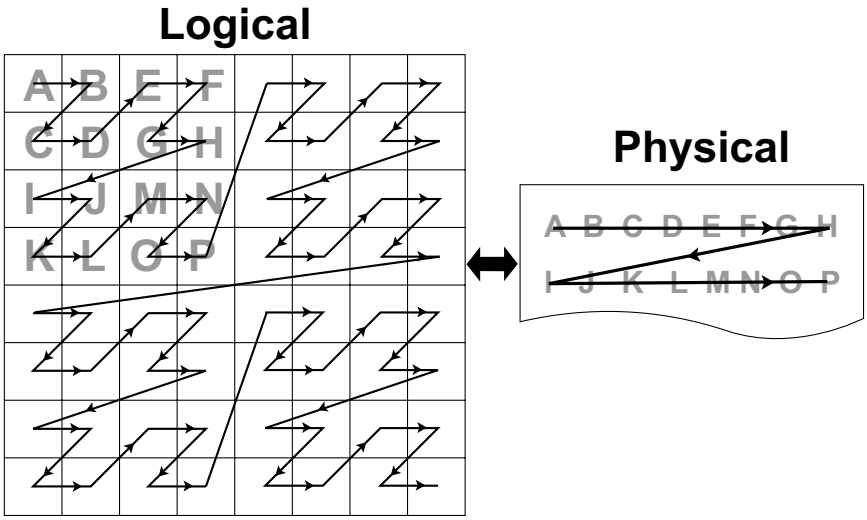

Figure 11 One Example of a Tiled Layout Format

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.

2.6.1 PCI Express Bus

Communication and data transfers between the system and the stream processor occur on the PCI Express® (PCIe®) channel. ATI 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.

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

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.

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 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.

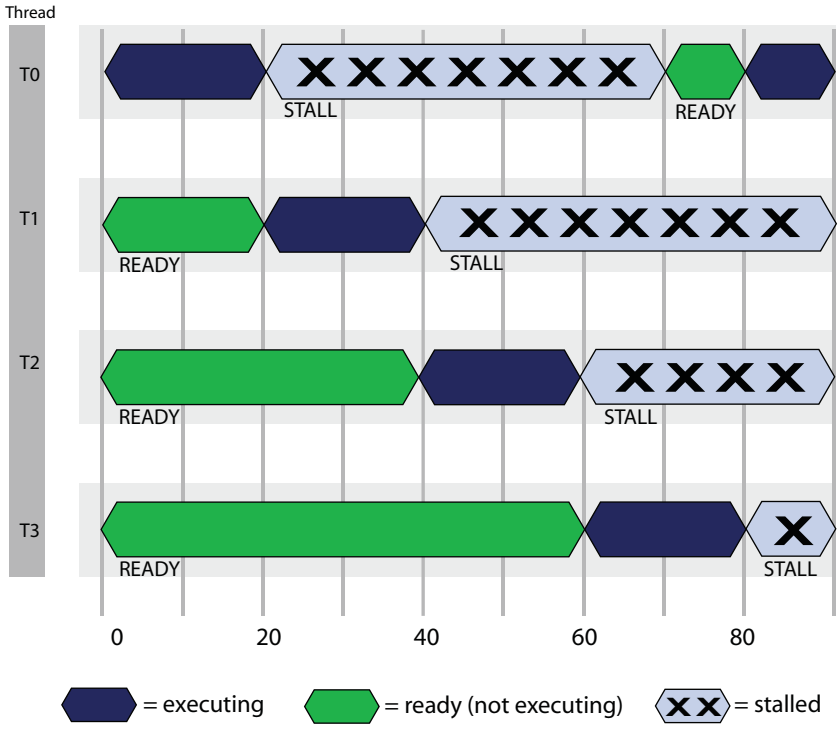

Figure 12 Simplified Execution Of Threads On A Single Thread Processor

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 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 13.

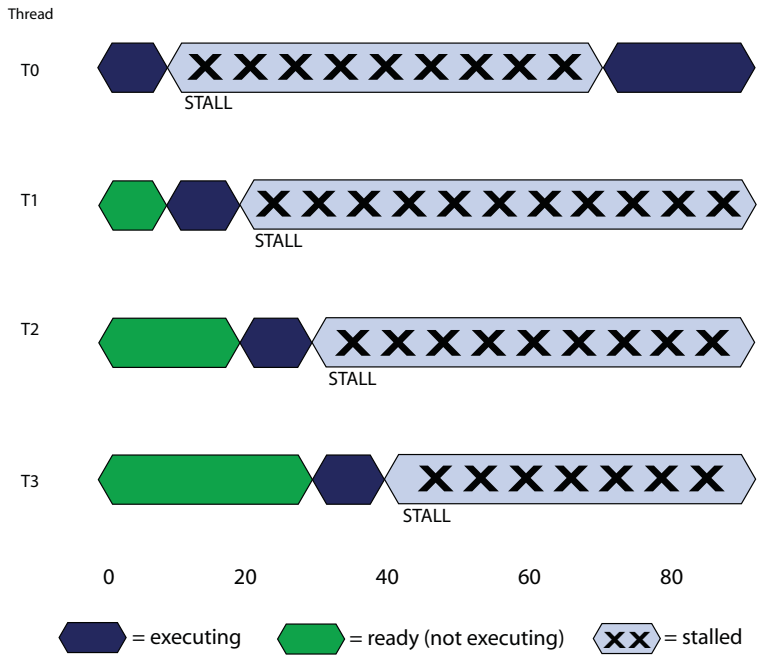

Figure 13 Thread Processor Stall Due to Data Dependency

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

3 Performance

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

3.1 Analyzing Stream Processor Kernels

Kernels must be compiled to native hardware instructions. The ATI Stream KernelAnalyzer (Figure 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 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 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.

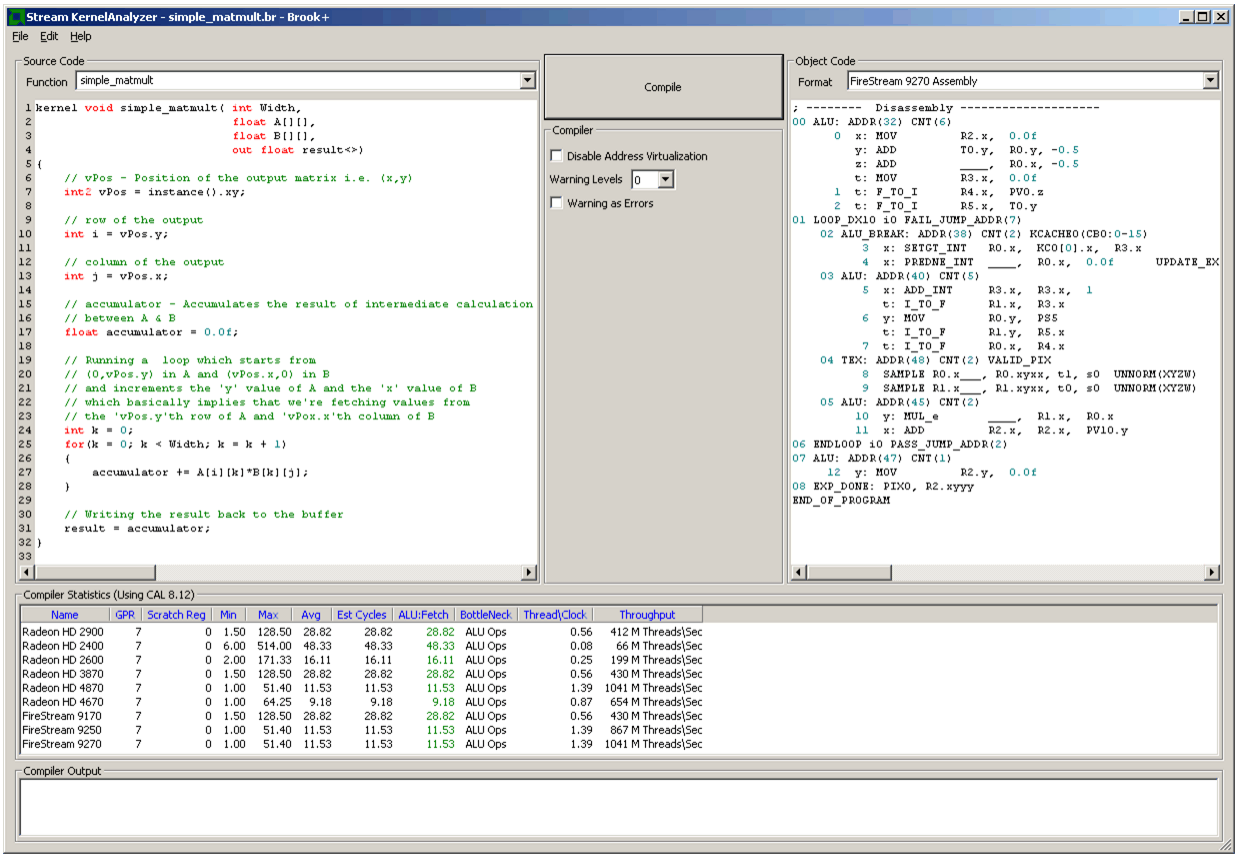

Figure 14 ATI Stream KernelAnalyzer Output

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 3.3, “Additional Performance Factors,” explores additional factors which affect performance.

3.3 Additional Performance Factors

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

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 Stream KernelAnalyzer 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.

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 2.2, “Thread Processing,” page 13, and Section 2.4, “Thread Creation,” page 14, 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.

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 3.2, “Estimating Performance,” page 21, 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.

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.

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.

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 2.4, “Thread Creation,” page 14) 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.

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.

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.

Contact

Advanced Micro Devices, Inc.
One AMD Place
P.O. Box 3453
Sunnyvale, CA, 94088-3453
Phone: +1.408.749.4000

For Stream Computing:
URL: **www.amd.com/stream**
Questions: **streamcomputing@amd.com**
Developing: **streamdeveloper@amd.com**
Forum: **www.amd.com/streamdevforum**