THE PEAKSTREAM PLATFORM:

High Productivity Software Development for Multi-Core Processors

PEAKSTREAM

EXECUTIVE SUMMARY

This paper discusses the PeakStream Platform, a new software development platform that offers an easy-to-use stream programming model for multi-core processors, including non-traditional processors such as Graphics Processing Units (GPUs), and the IBM CELL processor. Although using processors such as GPUs can provide dramatic performance advantages for High Performance Computing (HPC), they can also present significant challenges for application developers. The PeakStream Platform overcomes those challenges to provide a developer-friendly and efficient interface for multi-core computing. This paper describes the application view of the PeakStream Platform and offers solutions to the challenges posed. Application code samples are provided, along with comparisons between stream programming and traditional serial programming.

1 Introduction

Since the beginning of modern computing, most computer programs have been written using a serial programming model. Apart from a brief era of parallel programming in the late '80s on systems such as Thinking Machines and MasPAR, serial programming has been the predominant programming model for more than 50 years. But with the availability of new, powerful data parallel processors, such as the GPU, as well as the imminent arrival of highly multi-core CPUs (possibly as many as 32 cores by 2010), the serial programming model faces significant sailing challenges . The emerging world of highly parallel systems requires a programming model that scales to the new generation of parallel architectures that are already coming onto the market.

The PeakStream Platform embodies a new parallel programming model that we call "stream programming." PeakStream solves the classic challenges of parallel programming by using data arrays for its core objects, and makes the task of distributing work across multiple cores tractable.

The PeakStream Platform is designed for computationally intensive applications and provides an easy-to-use abstraction that is efficient, addresses the implementation details of various parallel architectures transparently to applications, and enables application portability between these architectures. So a PeakStream program written to run on today's GPUs, for example, will run seamlessly on the highly multi-core Central Processing Units (CPU's) or Cell processors of the future, without rewriting or recompilation.

Developers can take advantage of their GPU or multi-core system by using the PeakStream C and C++ Application Programming Interfaces (APIs). Those APIs are implemented by libraries that dynamically translate the API calls VM into parallel programs and execute them. These parallel programs are optimized to generate fast, numerically accurate code for the target platform. The PeakStream Platform also includes debugging support and profiling tools. In addition, it has been designed to interoperate well with developers' existing software development tools.

In summary, the PeakStream Platform is designed for computationally intensive applications that want to take advantage of the impressive performance of the new generation of multi-core systems. Today GPU's are the most highly parallel commodity processors currently on the market, offering as much as 10x the floating point performance of the best CPU on the market. This paper focuses on the PeakStream model for stream programming on GPUs.

1.1 Overview of this Paper

The performance of multi-core processors, as well as the set of issues that makes them difficult to program, is discussed in Section 2, with particular reference to GPUs.

The PeakStream Platform is introduced and examined in three sections. In Section 3, an application-level view of the platform is provided, along with a discussion of some operational details. Section 4 discusses the ways in which the PeakStream Platform addresses the challenges of multi-core programming. A larger code sample is shown in Section 5, and its performance is compared to that of a serial implementation of the same algorithm.

Section 6 relates the PeakStream Platform to other work that has been done to enable higher efficiency and higher productivity programming on multi-core processors.

Finally, Section 7 concludes the paper.

2 Multi-Core Processors

The best known multi-core processors are GPUs and the IBM Cell Processor [Flachs05][Pham05]. Although Intel and AMD are shipping multi-core processors, these are only dual-core and lack the same level of challenges that both GPUs (48 cores) and CELL(8 cores) show. So most of this paper focuses on GPUs, but where appropriate, we will make broad statements that are applicable to the Cell Processor as well. This paper does not cover the additional challenges that multi-core CPU programmers will face in the near future as the number of cores per CPU accelerates.

2.1 Traditional CPU Performance and Growth Rate

Traditional "single-core" CPU performance hit a fundamental performance limitation in early 2000 due to power and implementation difficulties. Until this wall was reached, applications could move seamlessly from one generation of processor to the next and take advantage of the increased clock speed without needing to change anything in their code. Now, however, the major CPU manufacturers have shifted to a multi-core strategy.

High-end commodity CPUs today are dual-core x86 processors. By the end of 2006, the first quad-core processor will ship, and by 2010, systems with more than 32 cores are projected to come to market.

Other computing platforms, however, have been multi-core for quite some time. Due to the inherently parallel task of rendering pixels, mainstream graphics processors have been designed as multi-core processors for decades. And in contrast to the CPU, GPUs today offer about 10x the floating point capacity of even a dual-core CPU.

2.1.1 GPU Performance and Architecture

Today, the CPU with the best floating point performance is the dual-core Intel Xeon 5160 which offers 48 gflops of single precision floating point performance. In contrast, the commodity high-end GPU offers 360 gflops of single precision floating-point performance and more than 50 GBytes/s of bandwidth to local memory. The computational performance growth rate for GPUs over the last few years has exceeded 2x per year. [Hanrahan05]

2.1.2 GPU and CELL Processor Systems

Full systems incorporating GPUs and CELL Processors involve several key architectural elements:

- A conventional CPU with one or more parallel execution threads
- Data-parallel computational units (fragment processors on the GPU and SIMD Processing Engines on CELL)
- Fast throughput-optimized external memory system
- Optional asynchronous DMA engine to accelerate data transfers between the processors

2.1.2.1 IBM Cell Processor System

The IBM Cell Processor is a single chip implementation of a multi-core system. [Pham05] The key architectural elements are embedded on a single Cell Processor chip:

- The PPU, an in-order CPU based on the IBM Power architecture
- Eight SPEs, each 4-way SIMD single precision floating point
- An asynchronous DMA engine to mediate transfers for these cases:
  - *between the PPU and SPEs*
  - *between the external memory and SPEs*
  - *between the eight individual SPEs*
- 25 GBytes/s of external memory bandwidth

The Cell Processor requires another interface chip in the system to support any I/O devices. A set of XDR memories is connected directly to the Cell Processor.

2.1.2.2 GPUs

Modern GPU systems provide a three-chip implementation. The three key architectural elements are:

- An x86 CPU in its own chip package with memory attached either locally or via a “Northbridge” chip.
- A “Southbridge” chip, connecting the CPU to the GPU, such as the NVIDIA nForce 4
- A programmable GPU with locally attached memory such as the ATI Radeon 1900 or NVIDIA GeForce 7900.

GPU characteristics such as the size and performance of the locally attached GPU and the number of computation cores varies depending on the GPU. For example, the ATI 1900XTX data-parallel computation units have the following characteristics:

- 48 processor cores
- Each core is 4-way SIMD
- Each core is capable of 3 single-precision flops/clock-cycle
- The compute processors operate at up to 625 MHz
- Over 50 GBs/sec of external memory bandwidth

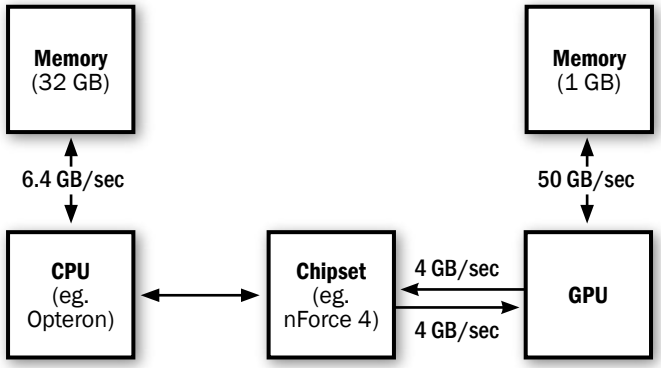

Figure 1: GPU System Diagram

Figure 1 illustrates the structure of a commodity x86 system with a high-end GPU, including the bandwidths of memory and I/O buses.

2.1.2.3 Processor Comparison

The following table compares the price, power, and performance of current processor configurations.

PROCESSOR COMPARISON			
	CPU	**GPU**	**Cell**
Processor	Dual-Core Xeon 5160	ATI 1900XT	IBM Cell Processor
Memory Capacity	32 GBytes	1 GBytes	512 MBytes
Memory Bandwidth	6 GBytes/s	50 GBytes/s	26 GBytes/s
Single-Precision GFlop/s	48	360	256

2.1.3 Use of GPUs on HPC algorithms

Performance of GPUs running hand-crafted code to implement HPC algorithms has been an area of substantial research. The following table shows recent results of running HPC applications that have been ported to GPUs. The GPU speedups are as reported by the paper authors compared to the CPU of the day.

1600 Bridge Parkway, Suite 102, Redwood City, Calif. 94065
www.peakstreaminc.com; info@peakstreaminc.com; 1-866-4PK-STRM (475-7876)

PERFORMANCE INCREASES	
Applications	**GPU speedup vs. CPU**
Navier-Stokes Fluid Flow [Scheidegger04]	16x
Black-Scholes Option Pricing [Kolb05]	15x
Gene Sequence Match (HMMer) [Horn05]	15x
Binomial Lattice Option Pricing [Kolb05]	10x
Large List Sorting [GOVINDARAJUM06]	10x
Lattice Boltzmann Fluid Flow [Ye04]	9x

The remainder of this paper focuses exclusively on GPUs as a case study for the PeakStream implementation of our stream programming platform.

2.2 GPU Programming Challenges

GPUs have a number of issues not addressed by any commercially available software environment.

2.2.1 Architecture changes

For HPC application developers, the most challenging factor in supporting GPUs is the variety and variability of those processors. GPU hardware is still actively evolving, and highly optimized implementations of algorithms on GPUs from year-to-year are often completely different. The differences between the Cell Processor and GPUs are also significant. Historically, a port to the GPU has not ensured an easy port to Cell, for example. The lack of portability has provided a significant barrier-to-entry for HPC application developers who understandably want to invest in a new application development model that will scale to multiple types of multi-core systems.

2.2.2 Poor tool support

GPUs have very limited tools for developing HPC applications. In particular, application-level profilers and debuggers do not exist.

For graphics programming in DirectX 9, Microsoft provides support for a debugger [Microsoft06] that can allow setting breakpoints and examining data in simulated GPU shaders. These simulated shaders operate entirely on the CPU, and simulate GPU shader computations. HPC algorithms (such as FFT) can easily translate into 20 or more GPU shaders for an algorithm. In addition, it does not expose performance or numerical issues that might only appear when running on real GPU hardware. Thus, debugging with simulated shader programs is inappropriate for HPC application developers.

GPUs do provide hardware support for performance monitoring, and some tools to access that information. [Domine05] Common GPU developer tools provide a visual display of performance information while rendering graphics application frames for interactive display. Unfortunately, they are directed at graphics applications rather than HPC applications. In particular, they require the use of OpenGL or DirectX, which is inappropriate for compute-based applications that have no interactive graphical display.

2.2.3 Arithmetic issues

GPU math libraries have been developed for use by interactive graphics applications, and in many cases suffer from substantial precision errors in basic math operations. For graphics applications, these precision errors are not noticeable since they result in transient single-pixel errors on the real-time display, effectively appearing as white noise. But for HPC applications, these errors are unacceptable since they jeopardize the quality of the underlying numerical simulation. In Figure 2, we present findings about exp precision on a modern GPU. Error is shown relative to a CPU implementation using IEEE 32-bit (single) floating point arithmetic. In the PeakStream VM, we provide an exp() function with more reasonable error characteristics for HPC applications.

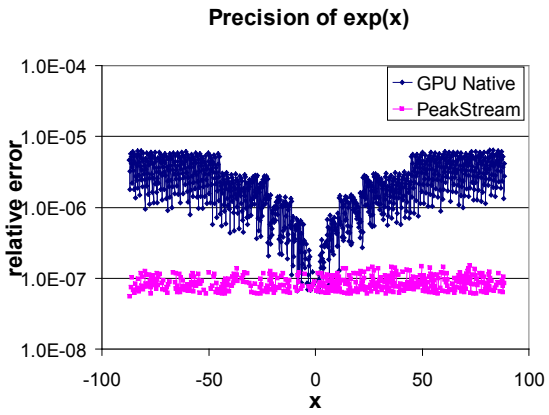

Figure 2: Precision of exp(x)

2.2.4 Need computationally intense kernels

In order to realize the performance potential of GPUs, it is critical to ensure that performance does not become memory-bandwidth limited. When examining the flop/memory-access ratio (also called computational intensity, [Brook04]) of GPUs, it is clear that GPUs have more floating-point units available for computation-per-memory-access than developers are used to on CPUs, as shown on the following table.

MEMORY COMPARISON		
	Xeon Dual-Core	**ATI R580**
Compute GFlop/s	48	360
Memory GBytes/s	6	50
Flop/(read+written bytes)	4.0	7.2

This is both a blessing and a curse. On the positive side, this provides high peak computational performance. On the negative side, it becomes critical on GPUs to structure computation in a way that ensures all these FLOPS are used effectively.

The following table computes the ideal FLOP/memory-access ratio for the ATI R580 for a simple operation with one input array and one output array:

MEMORY COMPARISON		
Compute performance	360	GFlop/s
Memory bandwidth	50	GBytes/s
Compute/memory transfer	7.2	flops/(read+written bytes)
Memory transfers/kernel	8	Bytes (2 floats) read+written/ kernel
Compute/kernel	57.6	flops/kernel

Thus, for the ATI R580 GPU, for a simple operation with one input and one output, the ideal computational intensity (assuming a linear memory access pattern) is 58. Kernels with ratios lower than this will have performance that is memory limited. Kernels with ratios higher than this will be compute limited. In general, the goal in developing kernels is to maximize the computational intensity. Naively coded kernels, by contrast, tend to be memory limited.

For a simple unary element-wise operation, like –x, this is unachievable. The way to generate dense computational kernels is to exploit producer-consumer locality and combine back-to-back element-wise operations. Exploitation of producer-consumer locality is one of the key characteristics of efficient algorithms for multi-core processors. [Dally03]

2.2.5 Loosely coupled processor

A GPU's data-parallel computation units operate asynchronously from the operation of the CPU with which they are coupled. For graphics applications, the GPU typically runs about one frame of latency behind the CPU. For an interactive visualization application, this means about 16 milliseconds of latency (16 msec = 1/60 of a second, which is a typical real-time rate of display). There are good reasons for GPUs to behave this way. In particular, it allows temporal load balancing of the command queue between the CPU and GPU. Graphics applications running on the CPU are typically very bursty, so the large (16 msec) command queue is included to avoid having the GPU "run dry" during periods when the CPU is not actively feeding commands to the GPU. [Akeley93]

2.2.5.1 High Latency

For our purposes, we view the GPU as a loosely coupled, high latency device. For good performance, the CPU must send a significant queue of work to the GPU for execution, and write it in a way that is latency-tolerant. This has led us to the design decision to disallow direct CPU access to data structures on the compute units on the GPU. Doing so would require that we operate the CPU and GPU in tight synchrony, which would destroy the performance advantages of the GPU.

2.2.5.2 High CPU/GPU Synchronization Costs

Reading data from the GPU back to the CPU is expensive. It requires that the GPU complete the computations that are queued for its execution which can often take tens of milliseconds, depending on how many commands are queued up for the GPU. Shortening the GPU command queue does not help because it just creates CPU/GPU temporal load balancing problems, as previously described. Synchronizing the CPU and GPU also has the undesirable consequence of forcing the GPU to momentarily "run dry," creating a window where the computational cycles on the GPU are completely wasted, until the CPU can queue some future commands for it.

To address this issue, we have structured the PeakStream Platform so that we can move an entire chain of computations to the GPU. By moving these computations to the GPU, the need to move data frequently between the processors and incur synchronization costs is reduced.

3 The PeakStream Platform

The PeakStream Platform is a comprehensive application development platform designed to maximize the floating point power of multi-core processors such as GPUs. It consists of four major components: the PeakStream APIs, the PeakStream VM, the PeakStream Profiler, and the PeakStream Debugger.

The structure of the PeakStream Platform is presented in this section, along with a description of several of the important features of the platform.

3.1 Structure of PeakStream Platform applications

Figure 3 shows the structure of an application interacting with the PeakStream Platform.

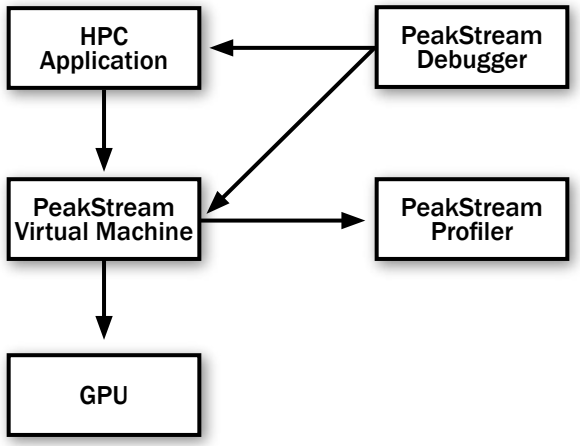

Figure 3: Application using the PeakStream Platform

The application is coded to use the PeakStream APIs, and linked against the PeakStream Virtual Machine (VM) libraries. The libraries handle all of the details of interaction with the processor. When the application uses the PeakStream APIs to perform mathematical operations, e.g., addition or use of math library functions like exp, those API calls are processed by the Virtual Machine. The Virtual Machine then creates optimized parallel kernels that are executed on the processor. The application must use explicit I/O calls (read and write) to move data into and out of the VM.

3.2 Array Data Types and Operations

The primary data type presented to the application developer by the PeakStream APIs is the Array. Arrays come in several flavors, most notably:

- Arrayf32: dense 32-bit (IEEE single) floating point elements.
- Arrayf64: dense 64-bit (IEEE double) floating point elements.

1600 Bridge Parkway, Suite 102, Redwood City, Calif. 94065
www.peakstreaminc.com; info@peakstreaminc.com; 1-866-4PK-STRM (475-7876)

Arrays can represent scalar (1x1), vector (Nx1), and 2-Dimensional or matrix (MxN) data. All operations provided by the PeakStream APIs operate on arrays. Static type checking is performed based on array types and dynamic checking and error reporting is used to signal errors such as array size mismatches or use of invalid array values.

All of the basic arithmetic operations, e.g., addition, multiplication, etc., can be performed on arrays. In most cases, these are performed element-wise, i.e., the corresponding elements of two arrays will be added or multiplied to produce a result array that has the same size as the inputs. Scalars are implicitly converted to 1D and 2-D arrays for use in element-wise operations, yielding semantics that are similar to those provided by Matlab and Fortran 90.

In addition to the basic arithmetic operations, PeakStream provides a substantial math library. It includes transcendental and trigonometric functions that operate on arrays, as well as matrix and vector reduction and manipulation functions. Advanced intrinsic functions including matrix multiplication, LU-decomposition, and convolution are also provided.

3.3 Just-in-Time / Dynamic Translation

As mentioned previously, applications use the PeakStream APIs to specify mathematical operations to be performed on arrays. The VM translates those operations dynamically into parallel programs, on a Just-in-Time (JIT) basis. Use of arrays as the fundamental data type in the PeakStream Platform, coupled with dynamic translation of programs, has the effect of decoupling the application programming model from the programming model of the processor being used. The VM has detailed knowledge of the specific processor being used (GPU/CELL/CPU). It performs optimizations necessary to make the application's array-based code perform well, providing results with adequate precision and accuracy.

Note that actual computation is decoupled from the application API calls to the VM. Computation is typically deferred until a computationally intense parallel kernel can be created, or until the application needs to read the computed data. Best performance will be achieved by applications that can perform lots of computation on array data between reads of the computed results.

3.4 PeakStream Platform Headers and Libraries

The PeakStream Platform supports use of the C and C++ languages for application development. Language bindings to platform operations are provided by a set of header files and shared libraries.

The goal of the PeakStream APIs is to enable developers the ability to express parallel computations in a natural, easy-to-use manner. As such, there are some differences between the C and C++ APIs. The two sets of APIs, however, can interoperate and can be used at the same time by an application.

3.4.1 "peakstream.h" Header File

There is one header for application use, "peakstream.h." It defines the PeakStream APIs for C and C++, specifically:

- The array datatypes and operations, as described previously,
- I/O operations, which move data into and out of PeakStream Arrays,
- Memory allocation interfaces, which allocate memory to be used for efficient I/O,
- Random number generation interfaces,
- Debugging and support interfaces, and
- Error handling interfaces.

The PeakStream C++ bindings, along with brief examples of their use, are presented in sections 3.5.

3.4.2 PeakStream Platform Libraries

A PeakStream application links against the PeakStream VM libraries. These are dynamically linked libraries, with a stable Application Binary Interface (ABI). This means that the application does not need to be recompiled or relinked when a new version of the PeakStream VM is released.

Taken together, the use of array data types for computation, the use of dynamic translation, and the use of a stable dynamically linked library ABI, present an interesting proposition for the application developer. Specifically, an application developer can port code once to the PeakStream Platform, and use it on whatever processors the Platform supports, including future generations of processors that did not exist when the application was written.

3.5 C++ Interfaces

The PeakStream C++ API is tailored to support common C++ idioms. Arrays are represented by C++ objects, and operator overloading is used extensively to support a style of programming familiar to C++ developers. Array memory is managed via object creation and destruction and leads to a very natural programming style.

3.5.1 C++ Interface Example

The following example shows a simple C++ code snippet that uses the PeakStream APIs to calculate the dot product of two vectors and return the resulting value. Dot product is calculated by multiplying the pairs of elements at the same positions in the two input arrays, then summing the result of those multiplications. The result is a single scalar value.

```
#include <peakstream.h>

using namespace SP;

Arrayf32 dot_product_cxx(const Arrayf32& a,
                         const Arrayf32& b)
{
  return sum(a * b, SP_ALL_DIMS);
}
```

In this example, the multiplication "a * b" is performed using an overloaded operator* that operates on PeakStream Array types. Likewise, the sum function is overloaded to support the array types.

Note that the result is returned as a PeakStream Array that contains a scalar value. and enables the result to be used efficiently by subsequent computations on the processor.

Also note that in this example, a temporary array is allocated and deallocated. The result of the multiplication "a * b" is allocated and deallocated automatically on behalf of the application by the C++ compiler.

3.6 PeakStream Platform Tools

The PeakStream Platform provides several tools for application developers. It includes an execution profiler and profile analyzer which can show where application cycles are being spent. Debugger interfaces are also provided and facilitate examination of PeakStream Arrays as they are being created and computed by the application.

3.6.1 Profiling Tools

The PeakStream VM can be configured to generate execution profiles while a PeakStream application is running. These execution profiles record the application's I/O to and from the GPU, GPU time consumed by the application or idle, CPU time used to generate parallel programs, etc. Resource consumption is attributed back to application source lines and to PeakStream API calls within each application source line.

An offline profile analysis tool can be used to generate gprof-style [FSF06] profiling reports from the saved profile data. The reports include application source code call points and the PeakStream API functions that are invoked, as well as per-function and cumulative resource consumption information. Most importantly, the profiler reports include data that can be used to identify two important GPU performance bottlenecks: excessive I/O, and parallel kernels that do not include enough computation to make full use of the processor. Once these bottlenecks are identified by the Profiler, the application developer can tune an application for optimal performance.

3.6.2 Debugging Interfaces

The PeakStream Platform includes several debugging interfaces that can be used to examine PeakStream Array data while debugging an application program. These are provided as scripts and functions and can be invoked from the debuggers supported by the PeakStream Platform.

Although debugger support for examining array data may seem straightforward, there are several issues that make doing so more complex when GPUs are involved.

First and foremost, the PeakStream Array data may not have been computed immediately when requested. GPUs are asynchronous, so there's no guarantee that the computation to generate the data will have completed when the debugger needs the data. In addition, the VM buffers computation to produce more computationally intense, parallel kernels, and so computation of the result might not even have been started when the result is requested. The VM must therefore ensure that the data is available when the debugger requests it.

Second, even if the array data has been computed, it may only exist in GPU memory. That memory is far away from the debugger that is actually debugging the application – typically across an I/O bus – that the debugger has no built-in facility to access. The VM must move data as necessary so that it is accessible to the debugger.

In addition to those two problems, the VM has to take care to not perturb normal system operation to supply values to the debugger. The debugger may request data from an array that would not even be computed during normal operation. For instance, computation of temporary values will normally be folded into other operations being performed on the GPU, and the temporary values will never be output. In that case, the VM must be able to provide the value that would be computed, while not disturbing the normal computations requested by the application.

4 Solutions to GPU Issues

As discussed earlier, application developers who want to harness the computational power of GPUs may have to work around a number of difficult issues. This section describes how the PeakStream Platform overcomes those technical challenges.

4.1 Software Stability when Processor Architecture Changes

First of all, because of its basic architecture, the PeakStream Platform insulates application developers from the architectural complexity of GPUs. Application developers never interact with the GPU directly, and instead use the APIs that are implemented by the PeakStream VM.

The PeakStream VM transforms application API calls so that they generate optimized, target-specific code for whatever multi-core processor is in use. This transformation can take into account optimal parallel kernel size, and can make use of optimized library routines.

Because PeakStream guarantees a stable ABI across multiple platform versions, applications can move to new versions of the PeakStream Platform without recompiling. So applications can ultimately use any processor hardware supported by the latest version of the VM without recompilation.

There is some overhead implicit in use of dynamic code generation. To keep this from adversely affecting application performance, the VM employs extensive caching so that commonly used code sequences can execute with very low overhead.

4.2 Tool Support and Compatibility

The PeakStream Platform provides important tools needed by application developers and verifies compatibility with third-party development tools.

As discussed in Section 3.6, the PeakStream Platform includes debugging tools that are geared toward HPC application development and profiling tools that can analyze the I/O and compute hot-spots related to the application's use of the GPU. These tools are the primary tools that an application developer needs to achieve high performance from any multi-core processor.

In addition, the PeakStream Platform has been designed to work with third-party tools and libraries. Industry-standard compilers, including gcc, Visual C++, and Intel CC are supported, as are the corresponding debuggers. Interoperability with communication libraries and other math libraries, e.g., common MPI libraries and the Intel Math Kernel Library (MKL), is also a feature of the PeakStream Platform. This allows developers to maintain their existing communication system and do piece-wise conversion of their applications to use the PeakStream APIs for computation. Finally, interoperability with software development and analysis tools such as CCov, gcov, and VTune [Intel03] has been designed into the PeakStream Platform so that application developers can find the hot spots in their code once compute is no longer their main cycle sink.

1600 Bridge Parkway, Suite 102, Redwood City, Calif. 94065
www.peakstreaminc.com; info@peakstreaminc.com; 1-866-4PK-STRM (475-7876)

4.3 Accurate Mathematical Library Support

As discussed in Section 2.2.3, native GPU math libraries may not support the accuracy necessary for use in HPC applications. Part of the work in porting the PeakStream VM to each new multi-core target is examining the native math library and creating optimized, accurate replacement math library routines as needed.

Figure 2 shows the accuracy of a typical GPU native math library function, along with the accuracy of the corresponding version created for the PeakStream VM. Virtually all of the GPU math library functions that were studied, including the transcendental and trigonometric functions, had similar accuracy issues. All were similarly improved for use in the PeakStream VM. Clearly, the ability of the PeakStream Platform to provide functions that work "out of the box" is of great benefit to the application developer.

4.4 Computationally intense GPU kernels

In order to be effective, GPU kernels need to have substantially greater computational intensity than traditional serial CPU code. In order to achieve this type of computational intensity when coding directly for a particular multi-core target, an application developer would have to analyze the application code carefully and break it into the right number of computational kernels. This optimization would have to stay within per-kernel limits on resource consumption and instruction count. These limits vary between different processors such as the ATI R580 and the IBM CELL. While this is possible, it is very time consuming, especially in the absence of an ecosystem of supported performance analysis tools. Further, it does not port well from one processor to another.

The PeakStream VM creates computationally intense parallel kernels automatically for the application developer. Application developers write simple, mathematical functions as described in the examples in Sections 3 and 5. When the application runs, the VM looks at the set of operations performed by the application. The PeakStream VM automatically fuses operations together into computationally intense parallel kernels. Because the VM does this dynamically, based on the processor in use, it can create optimal kernels for whatever processor is available. [Chan05][Riffel04]

4.5 Progressive Evaluation

The PeakStream VM hides the latency inherent in the loosely coupled GPU system architecture by allowing the application to continue while computation is being performed by the GPU. As discussed in Section 3.3, compilation and execution of parallel kernels is decoupled from application API calls. That is, the application may make an API call to add two arrays, and this will be executed some time later, just in time to return the needed result data back to the application. We call this feature of the platform "progressive evaluation".

Progressive evaluation allows for latency hiding. The application can issue a large number of compute requests into the VM and can continue processing other application work while the VM is performing the computations. That application work may include I/O (e.g., talking to other nodes in a compute cluster), or may include the generation of more compute work for the VM to process later.

To exploit the multiple CPUs common in today's HPC systems, the VM employs multiple threads to handle compilation, execution, and data movement. These threads allow the application to continue its work – while in parallel – the VM handles compilation of programs and asynchronous communication with the stream processor.

4.6 I/O cost analysis

Not all costs related to I/O with a GPU can be eliminated. But to achieve best performance, movement of data between the CPU and the GPU must be kept to a minimum. The PeakStream Platform encourages careful management of data movement in two ways: explicit interfaces and I/O analysis tools.

The PeakStream APIs include explicit data movement interfaces that move data between application arrays and PeakStream Arrays. The write operation copies application data into a PeakStream Array where it may be used for GPU computation. Similarly, the read operation reads data out of a PeakStream Array into an application array. Both read and write are capable of performing common scatter/gather operations to allow application flexibility in the way that they manage their data. Use of explicit APIs for I/O encourages the application developer to be cognizant of the I/O being done by the application.

As mentioned in Section 3.6.1, the PeakStream Profiler includes the ability to analyze application I/O to the GPU. Using the Profiler, an application developer can very quickly identify GPU I/O in the program, but more importantly, he can prioritize it so that performance issues due to excessive I/O can be quickly resolved.

5 Examples of using the PeakStream APIs

A primary advantage of using the PeakStream Platform is that it enhances developer productivity and reduces time-to-solution for solving computationally intensive problems such as monte carlo simulations of fixed income derivatives on Wall Street, or seismic migration within the energy sector. There is a growing realization in the HPC development community that improving developer productivity is increasingly important. [Kepner04] In this section, we walk through a sample application written in a serial fashion compared to an implementation using the PeakStream Platform.

5.1 Monte Carlo options pricing source code

Monte Carlo simulation is often used in financial markets to calculate the price of the stock option. The following source code sample shows an example of a traditional serialized code using the Intel MKL library [Intel04].

```
float sum1      = 0.0;
float deltat    = T/N;
float muDeltat  =
      (rate-div-0.5*vol*vol)*deltat;
float volSqrtDeltat = vol*sqrt(deltat);
VSLStreamStatePtr stream;
float *deviate = new float[M];
float *tmp     = new float[M];
vslNewStream(&stream, VSL_BRNG_MRG32K3A,
             1);

for(int j=0; j<M; j++)
   tmp[j] = 0.0f;
```

```
for(int i=0; i<N; i++) {
   vsRngGaussian(
            VSL_METHOD_SGAUSSIAN_BOXMULLER2,
            stream, M, deviate, 0.0f, 1.0f );
    for(int j=0; j<M; j++) {
       tmp[j] += deviate[j];
    }
}
sum1 = 0.0;
float factr = log(price) + N * muDeltat;
for (j = 0; j<M; j++ ) {
   float lnS1 = factr + volSqrtDeltat
                * (tmp[j]);
   float lnS2 = factr + volSqrtDeltat
                * (-tmp[j]);
   float S1 = exp(lnS1);
   float S2 = exp(lnS2);
   sum1 += 0.5*(max(0,S1-strike)
           + max(0,S2-strike));
}
return (exp(-rate*T)*sum1/M);
```

5.2 Monte Carlo options pricing using stream programming

The following source code sample shows the same algorithm using the PeakStream stream programming model. Note that the PeakStream version is simple to read and understand.

```
float deltat   = T/N;
float muDeltat =
      (rate-div-0.5*vol*vol)*deltat;
float volSqrtDeltat = vol*sqrt(deltat);

Arrayf32 meanSP; // result
{
   RNGf32 rng_hndl(SP_RNG_CEICG12M6, 0);

   Arrayf32 U = Arrayf32::zeros( M );
   for(int i=0; i<N; i ++) {
      U += rng_normal_make(rng_hndl, M);
   }
   Arrayf32 values;
   {
   float factr = log(price) + N * muDeltat;
   Arrayf32 lnS1 = factr + volSqrtDeltat*U;
   Arrayf32 lnS2 = factr + volSqrtDeltat*(-U);
   Arrayf32 S1 = exp(lnS1);
   Arrayf32 S2 = exp(lnS2);
   values = (0.5 * (max( 0,S1-strike )
      + max( 0, S2-strike) ) * exp( -rate*T ));
   }
   meanSP = mean( values );
}
return(meanSP.read_scalar());
```

6 Related work

There is no standard, universally accepted programming approach for multi-core processors. On CPUs, OpenMP is often used as a parallel approach, but it is recognized as being an overly complex programming model that is limited to thread-parallel processors. In addition:

- Because using multi-core processors such as GPUs for computation is relatively new, their programming tools are correspondingly immature.
- GPU and CELL programming approaches have been application-specific, targeting graphics, multimedia, or signal processing.
- GPU programming languages also have been architecture-specific. The architectures of GPUs are very diverse, both in the types of the parallelism they support and in their memory hierarchies. Each typically supports multiple forms of parallelism, including traditional SIMD, 4-way SIMD instructions large-scale and small-scale multi-threading, superscalar, and VLIW. Memory systems of GPUs and CELL offer different combinations and configurations of direct access to shared memory, DMA access to shared memory, local memories, caches, and long and short vector registers.

6.1 Commercial GPU Shader Languages

Graphics developers program modern GPUs with either OpenGL [OpenGL05] or Microsoft Direct3D [DX9SDK] APIs, using software drivers provided by the GPU vendors. OpenGL and Direct3D refer to GPU programs as shaders. These APIs are very similar, and provide two fundamental pieces of functionality, an API for compiling and executing shader programs and an API for traditional graphics functionality, such as rendering triangles, points, and quadrilaterals, loading texture maps, etc. Shader programs are executed on each pixel as triangle rendering generates them for subsequent display.

Shader programs may either be written in C-like languages, such as Cg [Mark03,Fernando03], HLSL [Peeper03], and GLSL [Rost04], or may be programmed with cross-platform pseudo-assembly languages, such as OpenGL ARB_Fragment_Program and Direct3D Shader Model Assembly. All of these shader languages share a number of common features:

- They provide an explicitly data-parallel programming model, not unlike MPL [Maspar91], where the developer manages parallelism and communication manually. This provides challenges for application developers.
- They are low-level languages that directly expose the features and limitations of the underlying hardware. They do not virtualize any hardware resources, so when a program would exceed any architectural limitations, such as the number of available registers or the maximum program size, it fails to compile. As such, code written for one GPU under these languages requires substantial rewriting to get all the benefits of another GPU.

6.2 GPU Software Research Efforts

Over the years, the research community has produced a number of interesting software projects targeting GPUs, some of which share common features with the PeakStream Platform.

None of these projects is widely used in industry, however, because they are not supported (or, in some cases, not available), or are not commercial quality in terms of features, robustness, and interoperability. They do not provide error-handling

1600 Bridge Parkway, Suite 102, Redwood City, Calif. 94065
www.peakstreaminc.com; info@peakstreaminc.com; 1-866-4PK-STRM (475-7876)

facilities, debugger and profiling support, numerically accurate implementations of transcendental functions, nor libraries of parallel numerical algorithms (e.g., matrix multiplication). Because the device interfaces and native instruction-set architectures of GPUs are not generally publicly available, they all use the aforementioned commercial GPU APIs and shading languages internally to execute on GPUs.

6.2.1 Sh

The pioneering Sh system from the University of Waterloo offers C and C++ APIs similar to the PeakStream APIs and was the first library to target GPUs using dynamic program generation. Not only does it primarily target graphics applications, but it provides little abstraction over the commercial shader languages and does not address most of the issues discussed in section 2 [McCool02,McCool04]. It is therefore deemed less suitable for use in HPC applications.

6.2.2 Brook

The Brook language from Stanford University is a highly restrictive, implicitly parallel C-like language that makes streams and element-wise computational kernels explicit. Brook was intended for multimedia processing and high-performance computing [Buck04]. Unfortunately, due to its restrictive programming paradigm, some communication-intensive algorithms that could achieve high performance on today's GPUs, such as LU decomposition, are difficult to express using Brook. Moreover, new languages rarely gain widespread acceptance because they force developers to learn the new paradigm and syntax and to use new compilers and related tools, which most developers consider undesirable.

6.3 Cell

Current commercial and open-source software for Cell simply provides basic compilers for its PowerPC and SPE cores and libraries to perform communication, synchronization, and basic program-management functions (e.g., starting programs) [CellProgDev]. Separate programs must be written for the PowerPC and SPEs, and spreading work across the eight SPEs is left entirely to the programmer. Even coping with the stringent local-memory limitations of the SPEs, for example, by loading code and data on-demand, must be done manually.

However, Cell provides a more conventional architecture than GPUs. It is essentially a tightly coupled distributed-memory multi-core processor with short-vector instructions similar to those implemented by CPUs [Flachs05]. Consequently, more traditional compiler technology can be applied to Cell. IBM Research has adapted IBM's XL product compiler to generate parallel programs for Cell from a single source program [Eichenberger05]. This compiler is not yet a product, however.

7 PeakStream: The New Generation of Multi-Core Systems

In this whitepaper, we have discussed the PeakStream Platform, a new software platform that provides an easy-to-adopt stream programming model for the new generation of multi-core systems. The PeakStream Platform implementation addresses many of the shortcomings of non-traditional processors such as the GPU through advanced portable VM technology and optimization techniques. We have shown how the PeakStream Platform addresses the limitations in programming multi-core systems by providing well-designed APIs, portability between processors, a commercial-quality mathematical library, and useful development tools.

References

[Allen06] ALLEN, E. et al., 2006. The Fortress Language Specification, version 0.866, Sun Microsystems, Inc., URL http://research.sun.com/projects/plrg/fortress0866.pdf

[Akeley93] AKELEY, K., 1993. Reality Engine graphics, Proceedings of the 20th annual conference on Computer graphics and interactive techniques, ACM, pp. 109-116.

[Buck04] BUCK, I. et al., 2004. Brook for GPUs: stream computing on graphics hardware, Proceedings of SIGGRAPH 2004, Computer Graphics Proceedings, Annual Conference Series, ACM, 23, 3, 777-786.

[Chamberlain04] CHAMBERLAIN, B., 2004. Chapel: The Cascade High Productivity Language, Language VMs '04, workshop paper and presentation, October.

[Chan02] CHAN, E. et al., 2002. Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association.

[Charles05] CHARLES, P. et al., 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing, Proceedings of the 20th Annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Annual Conference Series, ACM, 519-538.

[DP05] Data-Parallelism, 2005. Technical Report MSR-TR-2005-184, Microsoft Research, December.

[DX9SDK] DirectX 9 SDK, Microsoft Corp.

[Domine05] DOMINE, S., 2005. OpenGL Performance Tools. URL http://download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_Performance_Tools.pdf

[Eichenberger05] EICHENBERGER, A. et al., 2005. "Optimizing Compiler for the Cell Processor",

PACT 2005.

[Fernando03] FERNANDO, R. AND KILGARD, M.J., 2003. The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics, Addison-Wesley.

[Flachs05] FLACHS, B. et al., Feb. 2005. A Streaming Processing Unit for a CELL Processor, International Solid-State Circuits Conference Technical Digest .

[FSF06] FREE SOFTWARE FOUNDATION. 2006. gdb User Manual. URL http://www.gnu.org/software/gdb/documentation

[FSF98] FREE SOFTWARE FOUNDATION. 1998. gprof Manual. URL http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html

[Gordon02] GORDON M., et al., 2002. A Stream Compiler for

Communication-Exposed Architectures. MIT/LCS Technical Memo LCS-TM-627.

[Govindarajum06] GOVINDARAJUM, N.K. et al., 2006. GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. Microsoft Research Technical Report (MSR-TR-2005-183), Accepted for publication in the Proceedings of the ACM SIGMOD International Conference on Management of Data. URL http://gamma.cs.unc.edu/GPUTERASORT/gputerasort_sigmod06.pdf

[Hanrahan06] HANRAHAN, P., 2006. Why is graphics hardware so fast?, PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming.

[Hennessy06] HENNESSY J. L., PATTERSON D. A., 2006. Compute Architecture: A Quantitative Approach, Morgan Kauffman.

[HPFF93] HIGH PERFORMANCE FORTRAN FORUM (HPFF). 1993 (July). High Performance Fortran Language Specification, Scientific Programming 2, 1, 1-170.

[Hillis89] HILLIS, W.D., 1989. The Connection Machine, The MIT Press, reprint edition.

[Hillis87] HILLIS, W.D. AND STEELE, J. G. L., 1987. Data parallel algorithms, Communications of the ACM 30, 1, 78.

[Horn05] HORN, D.R. et al., 2005. ClawHMMER: A Streaming HMMer-Search Implementation, Proceedings of the ACM/IEEE SC2005 Conference on Supercomputing. URL http://graphics.stanford.edu/papers/clawhmmer/hmmer.pdf

[Intel03] INTEL CORPORATION, 2003. Technologies for Measuring Software Performance (White paper 253445-001). URL http://cache-www.intel.com/cd/00/00/21/93/219307_measuring_performance.pdf

[Intel04] INTEL CORPORATION, 2004. Intel math kernel library. URL http://www.intel.com/software/products/mkl.

[Kepner04] KEPNER, J., AND KOESTNER, D., 2004. HPCS Application Analysis and Assessment. http://www.highproductivity.org/kepner HPCS.htm

[Kolb05] KOLB, C. AND PHARR, M., 2005. Options pricing on the GPU, GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation, Fernando, R. and Pharr, M. eds., Addison Wesley Professional.

[Labonte04] LABONTE, F. et al. 2004. The Stream Virtual Machine, Proceedings of 2004 International Conference on Parallel Architectures and Compilation Technique, PACT '04, 267-277.

[Litvinov04] LITVINOV, V., 2004. Streaming Virtual Machine and Two-Level Compilation Model for Streaming Architectures and Languages, Language VMs '04 workshop paper and presentation, October.

[Mark03] MARK, W. et al., 2003. Cg: A system for programming graphics hardware in a C-like language, Proceedings of SIGGRAPH 2003, Computer Graphics Proceedings, Annual Conference Series, ACM, 22, 3, 896-907.

[Maspar91] MASPAR COMPUTER CORPORATION, 1991. MasPar Fortran Reference Manual.

[McCool04] MCCOOL, M. AND DU TOIT, S., 2004. Metaprogramming GPUs with Sh, AK Peters, Ltd.

[McCool02] MCCOOL, M., ZHENG Q., AND POPA T.S., 2002. Shader Metaprogramming, SIGGRAPH/Eurographics Graphics Hardware Workshop, Saarbruecken, Germany, September 2-3, 57-68.

[Microsoft06] April 2006. URL http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/Shader_Debugger.asp

[OpenGL05] OpenGL Architecture Review Board, 2005. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2 (5th Edition) Addison-Wesley Professional.

[Peeper03] PEEPER, C. AND MITCHELL, J.L., 2003. Introduction to the DirectX® 9 high level shading language, ShaderX2 - Introduction and Tutorials with DirectX 9.0, Engel, W. F. ed., Wordware, September.

[Peercy00] PEERCY, M.S. et al., 2000. Interactive multi-pass programmable shading, Proceedings of SIGGRAPH 2000, Computer Graphics Proceedings, Annual Conference Series, ACM, 19, 3, 425-432.

[Pham05] PHAM, D. et al., 2005 (Feb). The Design and Implementation of a First-Generation CELL Processor, International Solid-State Circuits Conference Technical Digest.

[Quinlan00] QUINLAN, D., 2000. ROSE: compiler support for object-oriented frameworks, Parallel Processing Letters, 10, (2,3), 215-226.

[Reynders96] REYNDERS, J. et. al., 1996. Pooma: A framework for scientific simulation on parallel architectures, Parallel Programming using C++, Wilson, G., and Lu, P., Eds., M.I.T. Press, 553-594.

[Riffel04] RIFFEL, A. et al., 2004. Mio: fast multipass partitioning via priority-based instruction scheduling, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, ACM Press.

[Rose87] ROSE, J. R., et al., May 1987. C*: An Extended C Language for Data Parallel Programming, Proceedings of the Second International Conference on Supercomputing, L. P. Kartashev et al., Eds., 2–16.

[Rost04] ROST, R., 2004. OpenGL Shading Language, Addison-Wesley Professional.

[Sankaralingam03] SANKARALINGAM, K. et al., 2003. "Exploiting ILP, TLP, and DLP Using Polymorphism in the TRIPS Architecture", 30th Annual International Symposium on Computer Architecture (ISCA), pp. 422-433.

[Sarkar04] SARKAR, V., 2004. X10: A Programming Model for Hierarchical Heterogeneous Parallelism, Language VMs '04 workshop paper and presentation, October.

[Scheidegger04] SCHEIDEGGER, C. et al., 2004. Navier-Stokes on Programmable Graphics Hardware Using SMAC, Proceedings of XVII SIBGRAPI - II SIACG 2004, 300-307. URL http://www.inf.ufrgs.br/~comba/papers/2004/smac.pdf

[Smith06] SMITH, A. et al., 2006, "Compiling for EDGE Architectures", 2006 International Conference on Code Generation and Optimization (CGO).

[Tarditi05] TARDITI, D., et al., 2005. Accelerator: Simplified Programming of Graphics Processing Units for General-Purpose Uses Via Data-Parallelism. URL http://research.microsoft.com/research/pubs/view.aspx?type=Technical%20Report&id=1040

[Taylor02] Taylor M. et al., 2002, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs", IEEE Micro vol 22, Issue 2.

[TMC86] THINKING MACHINES CORPORATION, 1986. C* Programming Manual.

[Veldhuizen98] VELDHUIZEN, T.L. AND GANNON, D., 1998. Active libraries: Rethinking the roles of compilers and libraries, Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientic and Engineering Computing (OO'98), SIAM Press.

[Ye04] YE, Z. et al., 2004. Lattice-Based Flow Simulation on GPU. URL http://www.cs.sunysb.edu/~vislab/projects/gpgpu/GPULBM.pdf

Essentials of PeakStream C++ Programming

peakstream.h
imports all PeakStream Classes

PeakStream classes & functions have their own namespace "SP::"

All data is expressed in the form of Arrays of 32 or 64 bit floating point numbers

Operator overloading converts C++ operators into data parallel operators

PeakStream API's are designed to look like common Intel MKL, Fortran, and Matlab

Stream arrays are automatically deallocated when they pass out of scope.

"make" and "write" functions move data into Stream Arrays for processing on the GPU

Stream arrays are opaque handles. Array data is copied back to system memory by using "read" calls

```
# include <peakstream.h>
using namespace SP;
...
...
int Conj_Grad_GPU_PS(int N, float *cpuA, float *cpux, float *cpub)
{
  int iter;
  Arrayf32 x = Arrayf32::zeros(N);
  {
      Arrayf32 A = Arrayf32::make2(N, N, cpuA);
      Arrayf32 b = Arrayf32::make1(N, cpub);

      Arrayf32 residuals = b - matmul(A, x) ;
      Arrayf32 p = residuals;
      Arrayf32 newRR = dot_product(residuals, residuals);

      for (iter = 0; iter < N; iter++) {
          Arrayf32 oldRR = newRR;
          Arrayf32 newX, newP, newResiduals;
          Arrayf32 Ap = matmul(A, p);
          Arrayf32 dp = dot_product(p, Ap);
          newX = x + p * oldRR / dp;
          newResiduals = residuals - Ap * oldRR / dp;
          newRR = dot_product(newResiduals, newResiduals);
          newP = newResiduals + p * newRR / oldRR;

          p = newP;
          residuals = newResiduals;

          float oldRRcpu = oldRR.read_scalar();
          if( oldRRcpu <= TOLERANCE) {
              break;
          }
          x = newX;
      }
  }

  x.read1(cpux, N * sizeof(float));
  return iter;
}
```