PEAKSTREAM

Supercomputing 2006
GPGPU Course
PeakStream Platform

Matthew Papakipos
CTO
PeakStream, Inc.

Talk Outline

SC06

- What’s wrong with GPGPU?
- The PeakStream Platform
  - Hardware requirements
  - Oil & Gas seismic demo
  - Application fit
  - PeakStream software architecture
- Concepts: Arrays, APIs, Kernel Synthesis
- Code Examples: Introduction, Seismic, Finance
- Application Developer Tools
  - Debugger & profiler demos
- GPGPU Gotchas & Tips
- Comparison with other platforms
- Future platform hardware & software directions

What's wrong with GPGPU?

- Programming interface is too low level
  - For applications programmers
  - Too vendor- and processor-specific
- Developer must determine appropriate stream kernels
  - But this changes with every GPU and hardware generation
- Poor tools for HPC applications
  - Debuggers are too low-level
  - Profilers are too oriented toward game graphics

PeakStream System Requirements

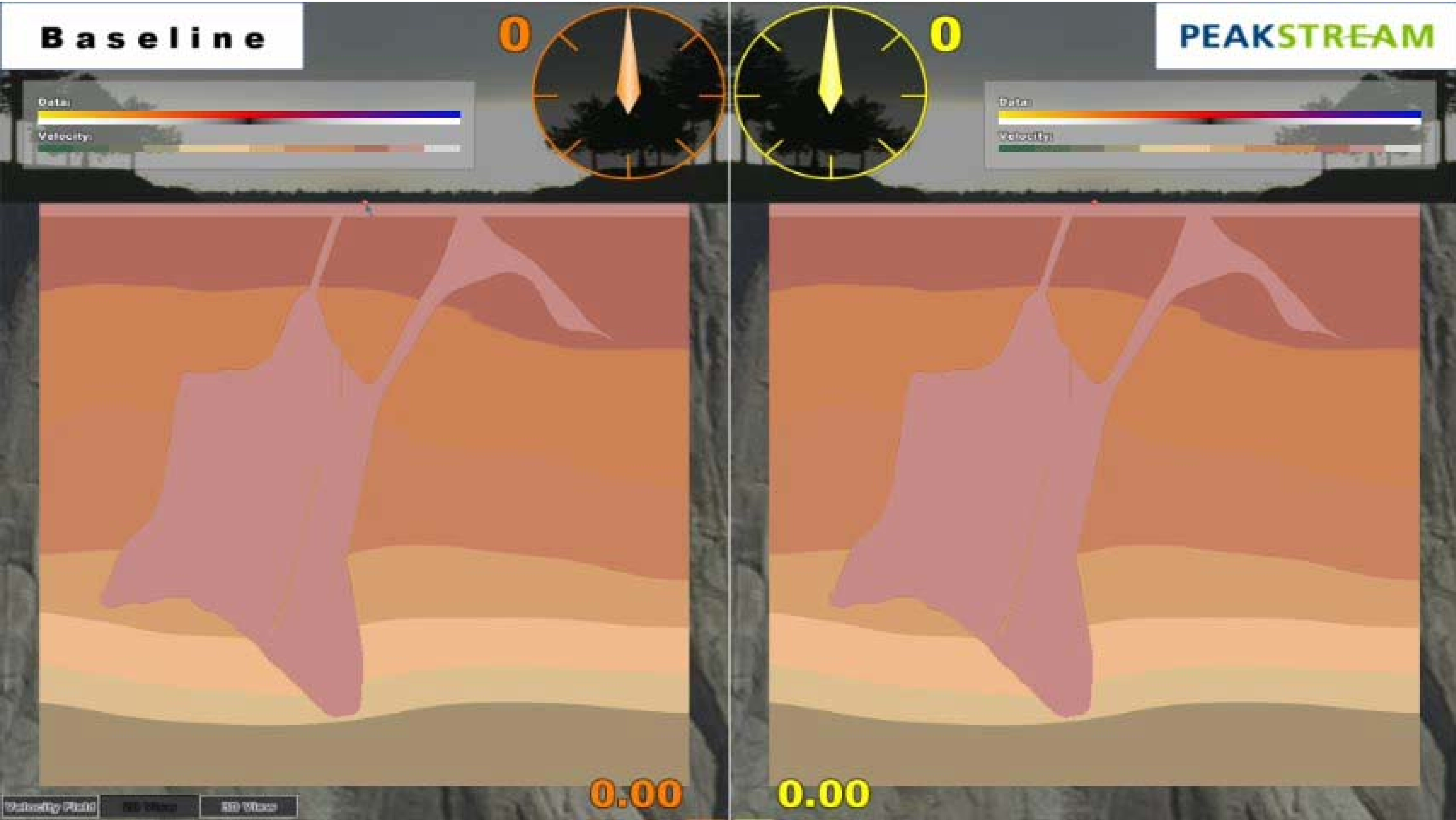
- HW Requirements
  - AMD or Intel CPU with SSE2/SSE3 support
  - 1 GB system memory
  - Optional ATI R580 GPU with at least 512 MB
- SW Requirements
  - Red Hat Enterprise Linux (RHEL) Enterprise Server, version 4.0, update 3
  - Compilers: gcc 3.4.5 or Intel Compiler 9.0
  - Debugger: gdb 6.3

Seismic Demo

- Seismic Acoustic Wave Demo
  - Acoustic wave simulation with constant modulus
  - Time discretization using 2nd order F-D method
  - Spatial discretization using 5x5 convolution
- Visualization Hardware
  - Windows 2GHz Opteron workstation, NVIDIA GPU
- Server hardware
  - Linux dual-socket, dual-core 2GHz Opteron, 3GB
  - CPU compute: Intel compiler, OpenMP, two threads
  - GPU compute: PeakStream with ATI R580 GPU 1GB
- 1 Gbit ethernet interconnect

SC06
PEAKSTREAM

Seismic Demo

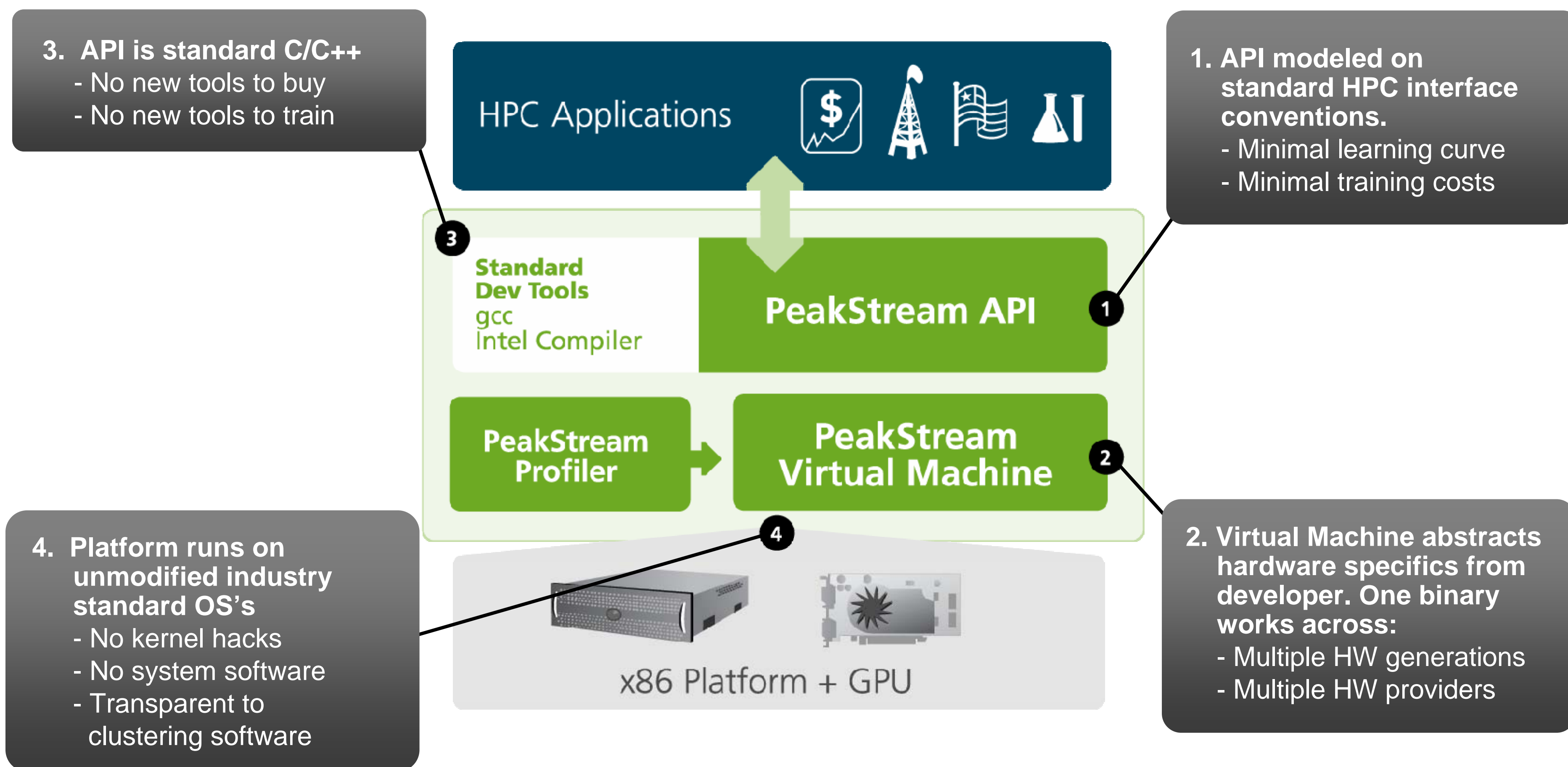


PEAKSTREAM

PeakStream Platform

- A commercial software platform
  - For developing and deploying HPC applications
  - Runs on multi-core CPUs
  - Supports GPU compute co-processors
- What applications are appropriate for PeakStream?
  - High flops counts
  - Large data-sets
  - High memory bandwidth requirements
  - Workload appropriate for the GPU

The PeakStream Platform™



Parallel Arrays

- Data Parallel
  - Predictable performance
  - Avoids load balancing problems with task parallelism
  - Long history in HPC hardware: Cray, CM, MasPar
  - Long history in HPC languages: APL, Fortran, Haskell
- Types
  - Arrayf32: computed on the GPU co-processor
  - Arrayf64: computed on the multi-core CPU

Data Parallel API

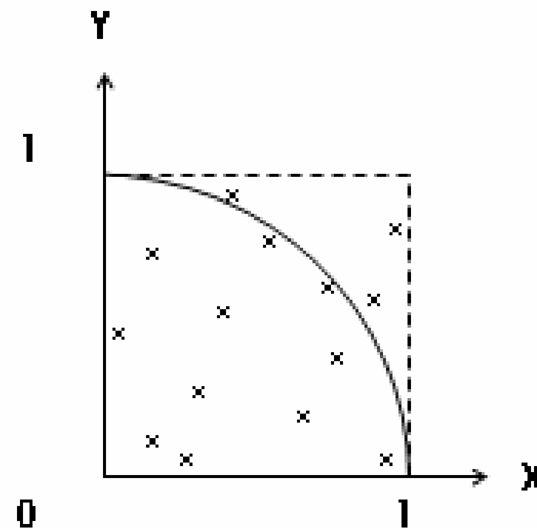
- All operate on Array objects:

- Element-wise: +, *, -, /, %, !, ||, ...
  - rounding, log/exp, trigonometry, distributions, ...
- Generators: identity, index, ones, zeros
- RNGs: Random Number Generators
- Reductions: sum, product, ...
- Indexing: sub-array extraction, gather, broadcast, ...
- Linear algebra
  - BLAS, Solvers
- FFT
- Convolution / Finite Difference
- Data Transfer: read & write
- Pragmas: read_hint, code reuse sections, ...

SC06
GPGPU
PEAKSTREAM

Sample Program: Compute π

- Generates a sequence of random numbers in the range [0,1)
- Every pair of random numbers determines a point (x, y)
- Approximates π as 4 times the ratio of points falling inside the circle $x^2 + y^2 <= 1$



GPGPU

PEAKSTREAM

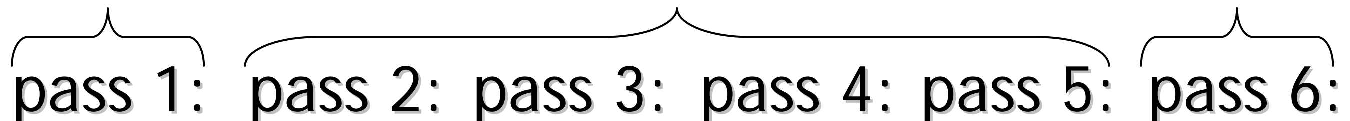
SC06

Computing π with PeakStream

```
#include <peakstream.h>

#define NSET 1000000            // number of monte carlo trials

Arrayf32 Pi = compute_pi();    // get the answer as a 1x1 array
float_pi = Pi.read_scalar();   // convert answer to a simple float
printf("Value of Pi  =  %f\n", pi);

Arrayf32
compute_pi(void)
{
   RNGf32 G(SP_RNG_DEFAULT, 271828);  // create an RNG
   Arrayf32 X = rng_uniform_make(G, NSET, 1, 0.0, 1.0);
   Arrayf32 Y = rng_uniform_make(G, NSET, 1, 0.0, 1.0);
   Arrayf32 distance_from_zero = sqrt(X * X + Y * Y);
   Arrayf32 inside_circle = (distance_from_zero <= 1.0f);
   return 4.0f * sum(inside_circle) / NSET;
}
```

Computing π with PeakStream

- This is the code the VM generates and runs:



Detail of pass 1
GPU code:

```
PS_OUTPUT main(float2 THR_ID : VPOS) {
    PS_OUTPUT output;
    float4 tmp0, tmp1, tmp2, tmp3, tmp4,
        tmp5, tmp6, tmp7, tmp8, tmp9,
        tmp10;
    tmp0 = CEICG12m6_1d(in0, THR_ID,
        inC0, inC1, inC2, inC3, inC4,
        inC5, out0_pad);
    tmp1 = smk32_mul(tmp0, inC6.x);
    tmp2 = smk32_add(tmp1, inC7.x);
    tmp3 = smk32_mul(tmp2, tmp2);
    tmp4 = CEICG12m6_1d(in0, THR_ID,
        inC8, inC9, inC10, inC11, inC12,
        inC13, out0_pad);
    tmp5 = smk32_mul(tmp4, inC14.x);
    tmp6 = smk32_add(tmp5, inC15.x);
    tmp7 = smk32_mul(tmp6, tmp6);
    tmp8 = smk32_add(tmp3, tmp7);
    tmp9 = smk32_sqrt(tmp8);
    tmp10 = smk32_le(tmp9, inC16.x);
    output.out0 = tmp10;
    return output;
}
```

SC06

PEAKSTREAM

Automatic Stream Kernel Synthesis

SC06

- Identifying the streaming kernel
  - What's the granularity of the inner loop?
  - How many GPU passes are optimal?
- It's inappropriate for the application to pick
  - It is *very* processor dependent
  - Depends on processor family, model, memory, ...
- This is a good task for compilers
  - This is what the PeakStream JIT compiler does
  - Ensures portability of your application code
  - Ensures scalable performance over many processors

PeakStream Debugger

- GDB debugger extensions to monitor PeakStream arrays
- Script provided for access

```
ps_gdb program
```

- DDE (Debugger Data Examination)

```
psprint array  (print contents of SP array)
SP::DDE::get_array_element(A, idx0, idx1)
SP::DDE::read1(A, outptr, size, stride)
SP::DDE::read2(A, outptr, size, stride, pad)
SP::DDE::write_array_to_file(A, filename)
```

PeakStream Analyzer

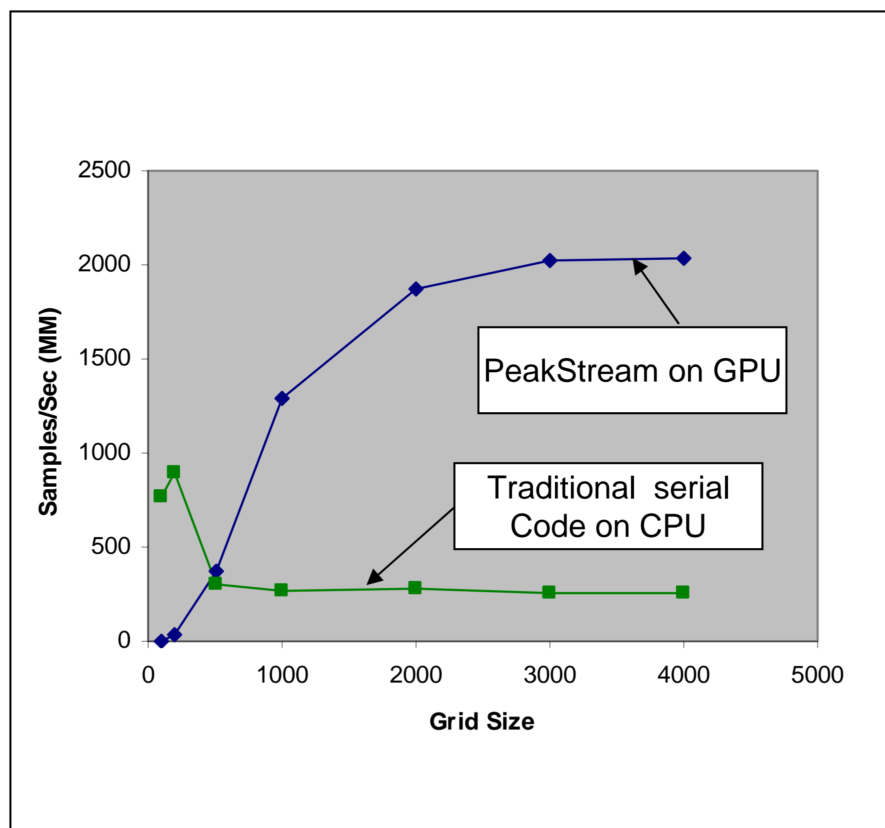
- A gprof-style application profiler.
- Usage: `ps_analyzer [options] [ > outfile ]`

```
 SP        %SP time  SP I/O  callee name                   file      line
 seconds             seconds
   0.299    100.00   0.00688

   0.114     38.05   0.00139 rng_uniform_make              main.cpp  142
   0.114     38.05   0.00139 rng_uniform_make              main.cpp  143
  0.0433     14.48         0 sum                           main.cpp  146
  0.0112      3.76  0.000137 sqrt                          main.cpp  144
  0.0112      3.76  0.000137 operator<=                    main.cpp  145
 0.00141      0.47  1.72e-05 operator*                     main.cpp  144
 0.00141      0.47  1.72e-05 operator*                     main.cpp  144
 0.00141      0.47  1.72e-05 operator+<...>                main.cpp  144
7.69e-05      0.03         0 operator/                     main.cpp  146
1.54e-05      0.01         0 operator*                     main.cpp  146
```

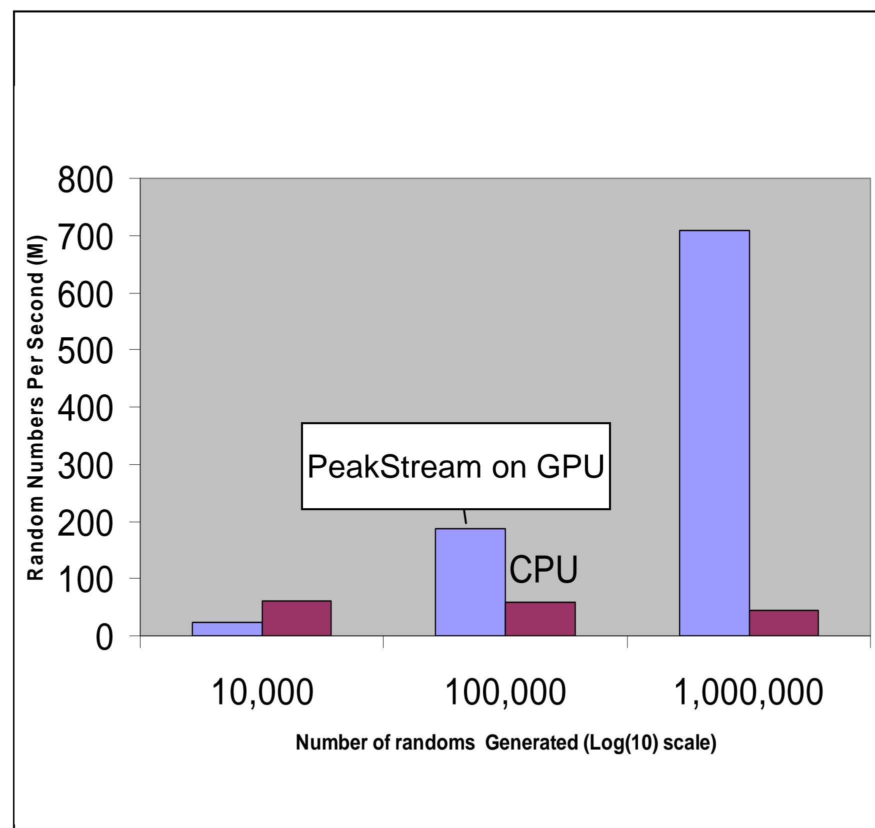
Lab Application Benchmarks

SC06

Oil & Gas: Kirchhoff Migration



Finance: Monte Carlo Simulation



8x Peak Performance Advantage

16x Peak Performance Advantage

GPGPU

PEAKSTREAM

Application: Kirchhoff Migration

```
void
KirchhoffMigration(int NT, int N, float *datagpu, float *modlgpu)
{
  int NTN = NT * N;
  float dx = LX / float(N);
  float dt = LT / float(NT);
  float factor = 1./ (velhalf * velhalf);
  float idt = 1./ dt;
  Arrayf32 modl = zeros_f32(NT,N);
  {
    Arrayf32 x = dx * index_f32(1, NT, N);
    Arrayf32 z = dt * index_f32(0, NT, N);
    Arrayf32 data = Arrayf32::make2(NT, N, datagpu);

    for(int iy=0; iy < N; iy++) {
      float y = float(iy)*dx;
      Arrayf32 index1 = float(iy) * ones_f32(NT, N);
      Arrayf32 it = 0.5 + sqrt( z * z + (x-y)* (x-y) * factor ) * idt;
      modl += gather2_floor(data, it, index1);
    }
  }
  modl.read1(modlgpu, NTN * sizeof(float) );
  return;
}
```

GPGPU

PEAKSTREAM

Application: Monte Carlo Finance

```
float MonteCarloAntithetic(float price, float strike, float vol,
                              float rate, float div, float T )
{
  float deltat       = T/N;
  float muDeltat     = (rate-div-0.5*vol*vol)*deltat;
  float volSqrtDeltat = vol*sqrt(deltat);
  float meanCPU       = 0.0f;
  Arrayf32 meanSP; // result
  {                // a new scope to hold temporary arrays
    RNGf32 rng_hndl(SP_RNG_CEICG12M6, 0);
    Arrayf32 U = zeros_f32( M );
    for(int i=0; i<N; i ++) {
      U += rng_normal_make(rng_hndl, M);
    }
    Arrayf32 values;
    {
      Arrayf32 lnS1 = log(price) + N * muDeltat + volSqrtDeltat*U;
      Arrayf32 lnS2 = log(price) + N * muDeltat + 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 );
  }                // all temporaries released as we exit scope
  meanCPU = meanSP.read_scalar();
  return meanCPU ;
}
```

SC06
PEAKSTREAM

GPGPU Gotchas & Tips

SC06

- Short vector performance
  - GPUs are best at long vectors
  - Beware function call overhead: use CRS API
- Make sure VM is not JIT-ing too much
  - Make sure VM code caches are effective
  - Use the PeakStream Analyzer to observe VM code cache hit rates in your application code
- Not supported by GPUs yet
  - Double precision
  - Integer data-types (but floats can often work)

Comparison with Other Systems

	Stream kernel synthesis	Language vs. API	Array Data-type	OS	Auto. SIMD-ization	HPC intrinsics (BLAS, etc.)	Debugger & Profiler
CG & HLSL	Manual	Language	No	Linux & Windows	Manual	None	Low-level, for games
Brook	Manual	Language	Yes	Linux & Windows	Manual	?	None
RapidMind	?	API	Yes	Linux & Windows	?	?	None
Accelerator	Auto	API	Yes	Windows	Manual	?	None
PeakStream	Auto	API	Yes	Linux & Windows	Auto	Included	High-level, for HPC apps

Future Hardware Directions

- Expect rapid convergence of
  - Cell
  - GPUs
  - Many-core CPUs

Convergent Processor Chip Characteristics:
- Many cores
- 4-way SIMD inside each core
- Types: Integer, Single, & Double
- Distributed memory (& NUMA)
- Multiple processors per system
- Volume commodity economics

- Double precision
- Integer data-types
- Distributed memories will be the norm
  - IDF: Intel 80-core prototype is not shared memory
  - using NUMA well requires treating it as distributed

Future Software Directions

- Windows support
- More processor vendors

- Double precision
- Integer data-types

- Focus on the HPC market

Key Technology:
- Familiar HPC array interface
- Automatic kernel generation
- Portability between processors
- Application debug & profile
- Workstation & Server