# The PeakStream Platform for Many-Core Computing

Matthew Papakipos
Engineering Director
Google

previously CTO
PeakStream, Inc.

# PeakStream History

» **PeakStream**
- Startup company
- Founded February 2005
- 35 people
- Based in silicon valley

» **PeakStream Mission Statement**
- Provide a software platform for High Performance Computing that unlocks the power of a new generation of processors, from GPUs to multi-core CPUs



PEAKSTREAM

# The PeakStream Team

- **Founder: Matthew Papakipos**
  - Former NVIDIA Director of GPU Architecture: NV20 & NV40 Lead, XBox
  - Graphics software standards: OpenGL & DirectX
  - Supercomputers: MasPar & Connection Machine
- **Chief Scientist: Pat Hanrahan**
  - Stanford computer science professor
  - Led the Brook project (more on this later)
- **Brian Grant**
  - Software architect, compiler expert
  - Formerly at Transmeta
- **Chris Demetriou**
  - Software architect, systems expert
  - Formerly at SiByte/Broadcom, NetBSD



PEAKSTREAM

# Google & PeakStream

» **PeakStream was acquired by Google in May, 2007**
  - Existing product line sales were discontinued
  - PeakStream's future is as part of Google

» **This presentation is a bit of history**
  - The founding of the PeakStream
  - The technology
  - The product
  - The Stanford connection



PEAKSTREAM

# Before PeakStream: Setting the Stage

- **The landscape before we founded the company**
  - GPUs had 10x the flops of CPUs: nv40 vs. pentium 4
  - Stanford had demonstrated the Brook project
  - Lots of buzz about “GPGPU”: What else can GPUs do?
- **Brook**
  - What was Brook?
  - Research developed in the Stanford Graphics Lab
    - Pat Hanrahan, Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Mike Houston, Kayvon Fatahalian
  - Demonstrated HPC codes running on GPUs
    - Using compiler technology to make it work
  - An open source project today

# Many-Core Processors

- **There is a large category of *Many-Core Processors***
  - GPUs: AMD & NVIDIA
  - IBM Cell Processor
  - Many-core CPUs: AMD & Intel
  - Future: AMD Fusion Processor = CPU+GPU Integration
- **Processor characteristics**
  - High memory bandwidth
  - Extremely high flops
  - High flop to memory access ratio
  - On-chip communication network
- **Why use many-core processors?**
  - Performance
  - Power
  - Cost

# Many-Core Processors

- **Are many-core processors new?**
  - No

- **Also called Stream Processors**
  - Imagine Processor, Bill Daly et al, Stanford
  - Merrimack Architecture, Bill Daly et al, Stanford
  - SPI, Chief Scientist: Bill Daly
- **GPU architecture was heavily influenced by Stream Processors**
  - As is the IBM Cell processor



PEAKSTREAM

# Who Wants All These FLOPs?

- **Gaming**
  - Physics
  - Image Processing
  - AI? This has not yet been demonstrated, but it's intriguing
- **Image Processing**
  - Image & Video Editing
  - Consumer & Professional
- **High Performance Computing**
  - Applications are solving big science problems numerically
  - Server compute farms: from 1,000s to 100,000s of CPUs
  - Workstations: CAD & Content. These have GPUs already
  - Embedded: Medical & Defense

# What is High Performance Computing?

- **HPC uses computation to solve a science problem**
  - Oil & Gas: Seismic analysis, reservoir modeling…
  - Finance: Monte carlo Simulations…
  - Biology: Molecular modeling, sequence matching…
  - Engineering: Fluid synamics…
  - Government Labs: Stockpile simulation, climate…
- **Who are HPC Developers?**
  - Mostly scientists, but not computer scientists
  - Mostly not parallel programming experts
  - Mostly like programming in MatLab
  - They are more interested in their science than in they are in optimizing a computer program

# What's Wrong with Multi-Core CPUs and GPUs?

» **Developer Productivity**
  - Most developers do not know how to write fast numerical codes
  - Making x86 run fast is hard.  GPUs are even harder.
  - Developing threaded applications is hard (OpenMP & pthreads)
  - Writing message-passing applications is very hard (MPI, Cell)

» **University curricula in numerical computing have shifted to high productivity languages**
  - MatLab: This is the tool of choice in hard sciences
  - Scientists no longer learn Fortran
  - Scientists are not computer scientists
  - Scientists are not parallel programming experts
  - Observation: MatLab is not a high performance system



PEAKSTREAM

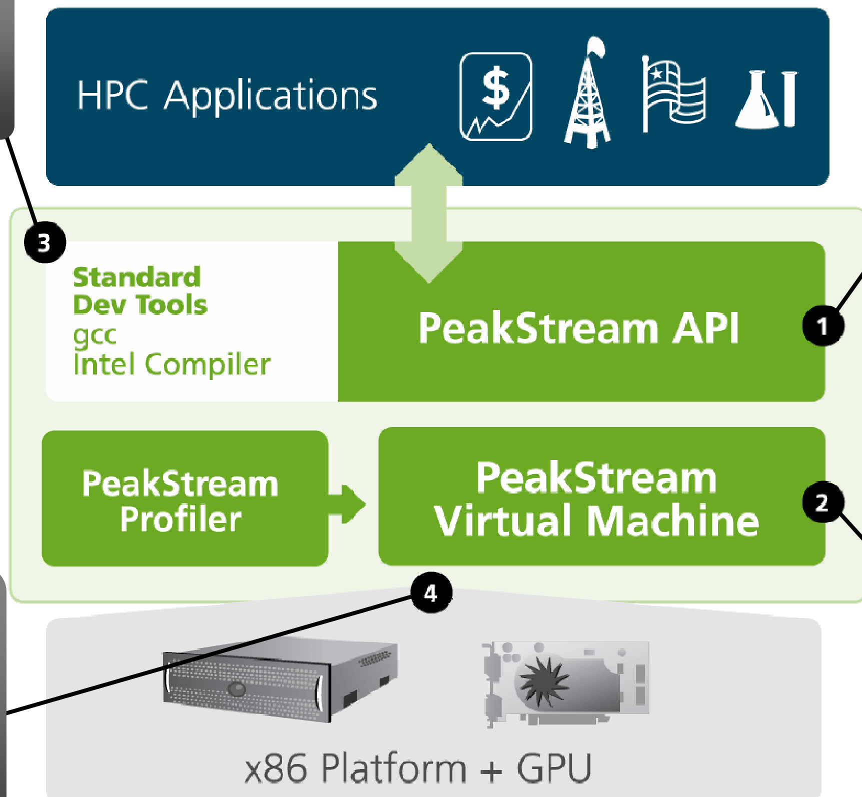
# The PeakStream Programming Model

- **We call it *Stream Programming***
  - A data-parallel programming model
  - With an explicit I/O model
  - For many-core processors
- **High performance**
  - The *raison d'etre*!
- **Portable**
  - Across processor vendors, across processor generations
  - (But does require significant effort by PeakStream)
- **Interoperable**
  - Leverage existing libraries, tools, and systems (MPI, gcc, etc.)
- **High productivity**
  - Minimize time to solution
  - For scientists & mathematicians
  - Tools are important: debugger & profiler

# The PeakStream Platform™

**3. API is standard C/C++**
- No new tools to buy
- No new tools to train

**1. API modeled on standard HPC interface conventions.**
- Minimal learning curve
- Minimal training costs




**4. Platform runs on unmodified industry standard OS's**
- No kernel hacks
- No system software
- Transparent to clustering software

**2. Virtual Machine abstracts hardware specifics from developer. One binary works across:**
- Multiple HW generations
- Multiple HW providers



PEAKSTREAM

# PeakStream Programming Essentials

Data expressed as Arrays of 32 or 64 bit floating point numbers

“make” and “write” functions move data onto the GPU for processing

Operator overloading converts operators into data parallel operators

Stream arrays are opaque. Data is copied back to system memory with “read” calls

APIs look like Intel MKL, Fortran, and Matlab functions

```
nt 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;
```

*Learnable in hours, proficient in days*

# Why an API?

- **New languages are rarely adopted**
  - They have steep learning curve
  - They require new software ecosystems
    - Compilers
    - Tools
    - Libraries
  - *Language extensions* are new languages
    - Definition of a new language: “won’t compile with an existing compiler”

- **APIs are much easier to adopt**
  - APIs are language-neutral
    - They allow people to use their favorite languages
    - They allow multiple language bindings: C, C++, Fortran, Java, ...
  - APIs facilitate interoperability with existing software ecosystems
    - MPI, OpenMP, MKL, ACML, ...
  - APIs and languages are equally expressive



PEAKSTREAM

# Virtual Machine with Dynamic Compilation

» **Dynamic compilation facilitates *binary* portability**

» **Across processor vendors**
- Dynamically compile and optimize for the processor at hand
- NVIDIA and ATI GPUs have totally different ISAs
- GPUs and CPUs have very different ISAs and OS interfaces

» **Across processor generations**
- Processors change faster than applications
- Want applications to automatically get faster as hardware gets faster
- But GPU ISAs change completely from one generation to the next
- Even x86 adds new instructions: SSE[1,2,3,4]



PEAKSTREAM

# Dynamic Compilation Well Suited for HPC

- **Dynamic compilation is now commonplace**
  - GPU drivers
  - Java and .NET
  - Transmeta
  - VMware, XenSource
- **Dynamic compilation is fast**
  - VMware running windows boots in 30 seconds
    - Just 1 second of that is JIT code translation
- **Code caching is tremendously effective for HPC**
  - Long running
  - Highly repetitive
- **JIT overhead easily amortized for HPC**
  - High data-to-code ratio



PEAKSTREAM

# 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;
}
```

# What the Compiler Does: Generate Compute Kernels

» **Compiler outputs a series of *Compute Kernels***
  - And the VM executes them on the processor

» **A compute kernel is structured as:**
  - Gather
  - Compute
  - Scatter

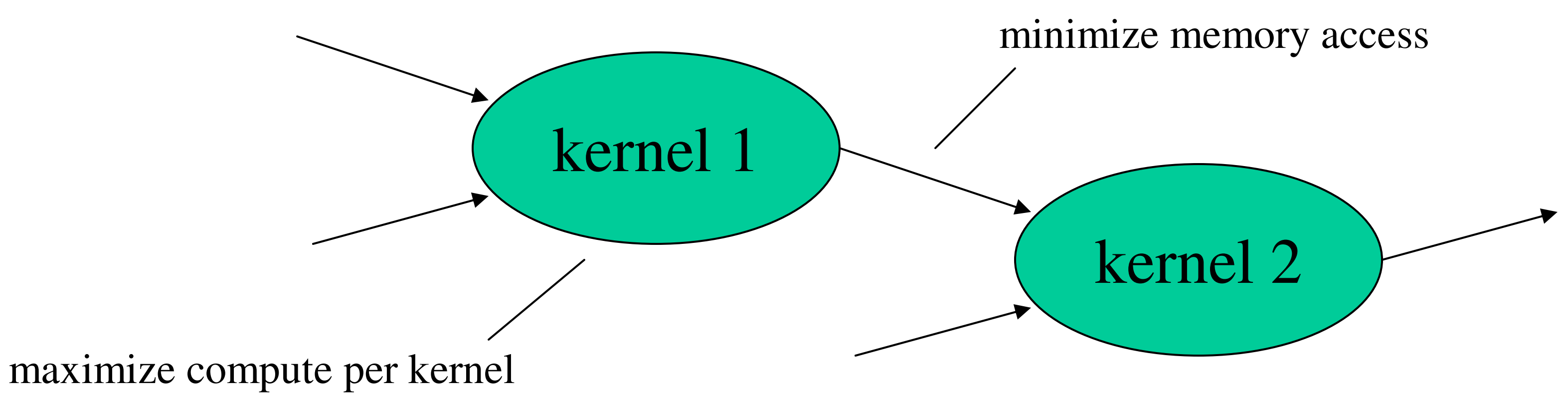
» **Maximize flops/kernel**
  - Minimize memory bandwidth requirements
  - Avoid the processor *memory wall*

» **All of this is done automatically by the PeakStream JIT Compiler**

# The Stream Programming Paradigm

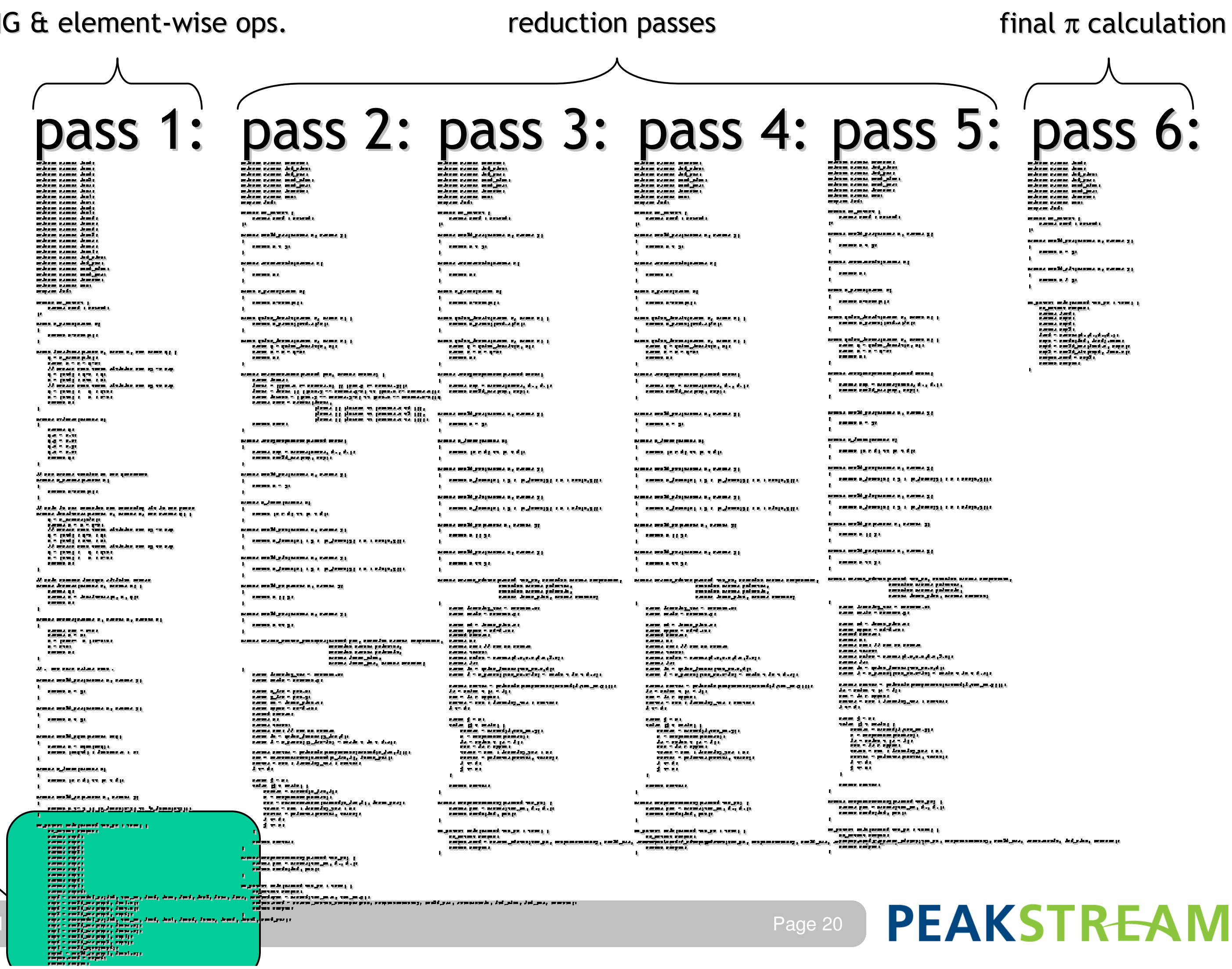
» **Computation expressed as composition of compute *kernels*:**
  - Gather phase
  - Compute phase
  - Scatter phase

» **Translates memory latency into memory bandwidth**
  - Able to exploit processors with high compute/memory access ratios



# 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;
}
```




PEAKSTREAM

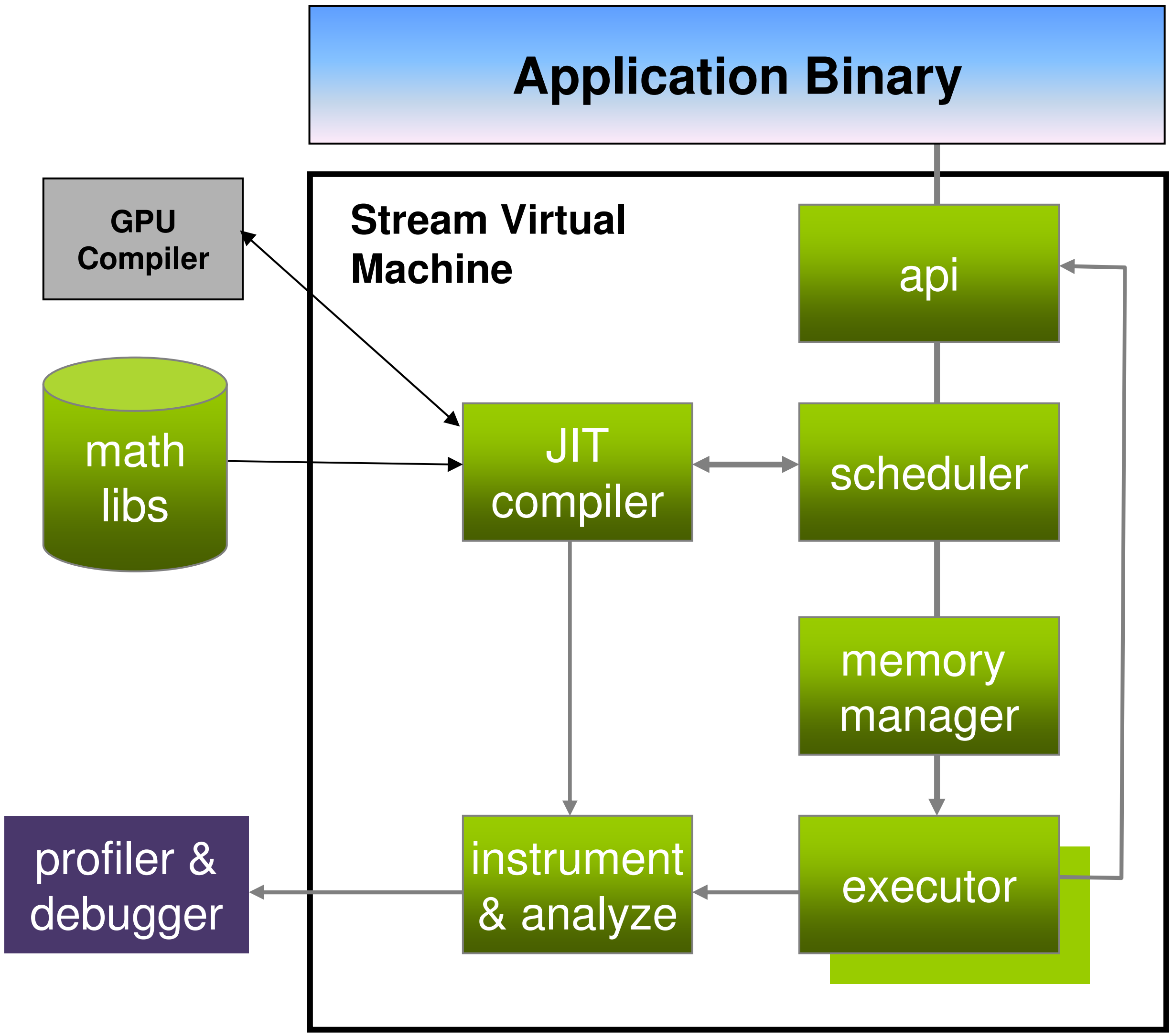
# Automatic Stream Kernel Synthesis

» **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 application code
  - Ensures scalable performance over many processors

# PeakStream Software Architecture



# PeakStream Platform Functionality

| | | |
|---|---|---|
| **Standard Math**<br>» Standard operators<br>» Range of logarithms<br>» Exp, powers, roots<br>» Rounding, abs etc. | **Trigonometry**<br>» Standard trigonometry<br>» Inverse trigonometry<br>» Hyperbolic functions | **Array Manipulation**<br>» Attribute queries<br>» Gather/Spread<br>» Indexing |
| **Array Reduction & Statistics**<br>» Sum/Min/Max<br>» Mean/Variance/Std Dev<br>» Random number generators | **BLAS**<br>» Full BLAS equivalance<br>» Levels 1/2/3 | **Signal processing**<br>» Convolution<br>» Multiple border options<br>» 1D, 2D FFTs |
| **Linear Algebra**<br>» Dot product/Transpose<br>» Matmul<br>» LU & Cholesky Solvers | **Array generation**<br>» Identity and zero arrays<br>» Random number arrays<br>» Data stride | **Utility functions**<br>» Data transfer<br>» Performance hints<br>» VM management<br>» Debug APIs |

| | |
|---|---|
| Platforms & language support | » Linux: Redhat Enterprise Linux, CentOS, gcc, gdb, icc<br>» Windows: WinXP SP2, Visual Studio, icc<br>» C/C++ |



PEAKSTREAM

# Mandelbrot Fractal

```
Arrayf32 Iter = -1;
Arrayf32 XP = (Arrayf32::index(0,pixels_x,pixels_y)-pixels_x/2);
Arrayf32 YP = (Arrayf32::index(1,pixels_x,pixels_y)-pixels_y/2);
Arrayf32 Xprime = XP*cos(phi) - YP*sin(phi);
Arrayf32 Yprime = YP*cos(phi) + XP*sin(phi);
XP = Xprime/(pixels_x*zoom)+cx;
YP = Yprime/(pixels_x*zoom)+cy;
for (int iteration=0; iteration<max_iter; iteration++)
{
    // Iterate
    Arrayf32 Y=2*X*Y+YP;
    Arrayf32 X=X*X-Y2+XP;
    Arrayf32 Y2=Y*Y;
    // Test for escape condition
    Arrayf32 Eval = cond(X*X+Y2<4,0,1);
    Iter = cond(Iter<0&&Eval>0,iteration,Iter);
}
```



PEAKSTREAM

# PeakStream Linux Tools Extensions

## Debugger: gdb Extensions

- Debug PeakStream applications with a plug-in to the standard gdb debugger
- Set breakpoints
- Step through code executing on GPU & CPU
- Examine arrays resident on the GPU
- Generate reference results to compare GPU execution to CPU execution
- Trap runtime errors

## Profiler: Collection and Analysis

- Insight into optimization potentials
- gprof style tool for analyzing application performance
- Shows time spent per line and per function
- Pinpoints excess data movement from system to local memory
- View stream processor compute kernels
- Analyze memory utilization



PEAKSTREAM

# 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)
```

- **Error handlers**
  - Either handle from your application or catch in the debugger
- **Generate reference results**
  - To compare GPU to CPU results
  - From your debugger session or your application



PEAKSTREAM

## PeakStream Profiler

- **A gprof-style application profiler**
- **Usage:**
  - `ps_analyzer [options] [ > outfile ]`
- **3 basic views**
  - API Call view
  - Compute kernel summary view
  - Compute kernel detail view

# Profiler: API Call View

| % Total Time Cumulative | % Total Time | Calls | Compute Time | I/O Time | VM Time | API Name | Caller Name | File | Line |
|---|---|---|---|---|---|---|---|---|---|
| 29.48 | 29.48 | 100 | 0.075 | 0 | 0.032 | SP::rng_uniform_make | GPU_PS_Compute_Pi | main.cpp | 142 |
| 58.94 | 29.45 | 100 | 0.075 | 0 | 0.032 | SP::rng_uniform_make | GPU_PS_Compute_Pi | main.cpp | 143 |
| 85.27 | 26.33 | 10 | 0.037 | 0 | 0.059 | SP::sum | GPU_PS_Compute_Pi | main.cpp | 146 |
| 90.75 | 5.48 | 10 | 0.014 | 0 | 0.0059 | SP::operator<= | GPU_PS_Compute_Pi | main.cpp | 145 |
| 95.54 | 4.79 | 10 | 0.012 | 0 | 0.0052 | SP::sqrt | GPU_PS_Compute_Pi | main.cpp | 144 |
| 96.35 | 0.81 | 10 | 0 | 0.0029 | 0 | SP::RNGf32::RNGf32 | GPU_PS_Compute_Pi | main.cpp | 140 |
| 97.06 | 0.71 | 10 | 0.00013 | 0 | 0.0025 | SP::operator/ | GPU_PS_Compute_Pi | main.cpp | 146 |
| 97.75 | 0.68 | 10 | 0.0017 | 0 | 0.00074 | SP::cond | GPU_PS_Compute_Pi | main.cpp | 145 |
| 98.43 | 0.68 | 10 | 0.0017 | 0 | 0.00074 | SP::operator+ | GPU_PS_Compute_Pi | main.cpp | 144 |
| 99.12 | 0.68 | 10 | 0.0017 | 0 | 0.00074 | SP::operator* | GPU_PS_Compute_Pi | main.cpp | 144 |
| 99.80 | 0.68 | 10 | 0.0017 | 0 | 0.00074 | SP::operator* | GPU_PS_Compute_Pi | main.cpp | 144 |
| 99.94 | 0.14 | 10 | 2.7e-05 | 0 | 0.00049 | SP::operator* | GPU_PS_Compute_Pi | main.cpp | 146 |
| 100.00 | 0.06 | 10 | 0 | 0.0002 | 0 | SP::Arrayf32::read_scalar | GPU_PS_Compute_Pi | main.cpp | 148 |

» **Role: report on how the application spent its time**
- in terms of PeakStream API calls

» **Conclusions for this simple example:**
- This application is not I/O limited
- Most of the run time is spent in the RNG & reduction



PEAKSTREAM

# Profiler: Compute Kernel Summary View

| % Total Time | Executions | Compute Time | Paging Time | JIT Time | Kernel Name | File | Line |
|---|---|---|---|---|---|---|---|
| 71.92 | 10 | 0.18 | 0 | 0.078 | GPU_PS_Compute_Pi:1 | main.cpp | 142 |
| 26.33 | 60 | 0.037 | 0 | 0.059 | GPU_PS_Compute_Pi:2 | main.cpp | 146 |
| 0.85 | 10 | 0.00016 | 0 | 0.0029 | GPU_PS_Compute_Pi:3 | main.cpp | 146 |

» **Role: report on which compute kernels matter most**

# Profiler: Compute Kernel Detail View

```
% Total  Executions  Compute   Paging    JIT    Kernel                    File             Line
Time                 Time      Time      Time   Name
  71.92          10     0.18         0   0.078  GPU_PS_Compute_Pi:1       main.cpp          142

        Details:

               Compute      JIT    API Name                     Caller               File        Line
               Time         Time

           0:     0.072      0.03   SP::rng_uniform_make         GPU_PS_Compute_Pi    main.cpp    142
           1:    0.0017   0.00074   SP::rng_uniform_make         GPU_PS_Compute_Pi    main.cpp    142
           2:    0.0017   0.00074   SP::rng_uniform_make         GPU_PS_Compute_Pi    main.cpp    142
           3:    0.0017   0.00074   SP::operator*                GPU_PS_Compute_Pi    main.cpp    144
           4:     0.072      0.03   SP::rng_uniform_make         GPU_PS_Compute_Pi    main.cpp    143
           5:    0.0017   0.00074   SP::rng_uniform_make         GPU_PS_Compute_Pi    main.cpp    143
           6:    0.0017   0.00074   SP::rng_uniform_make         GPU_PS_Compute_Pi    main.cpp    143
           7:    0.0017   0.00074   SP::operator*                GPU_PS_Compute_Pi    main.cpp    144
           8:    0.0017   0.00074   SP::operator+                GPU_PS_Compute_Pi    main.cpp    144
           9:     0.012    0.0052   SP::sqrt                     GPU_PS_Compute_Pi    main.cpp    144
          10:     0.014    0.0059   SP::operator<=               GPU_PS_Compute_Pi    main.cpp    145
          11:    0.0017   0.00074   SP::cond                     GPU_PS_Compute_Pi    main.cpp    145
```

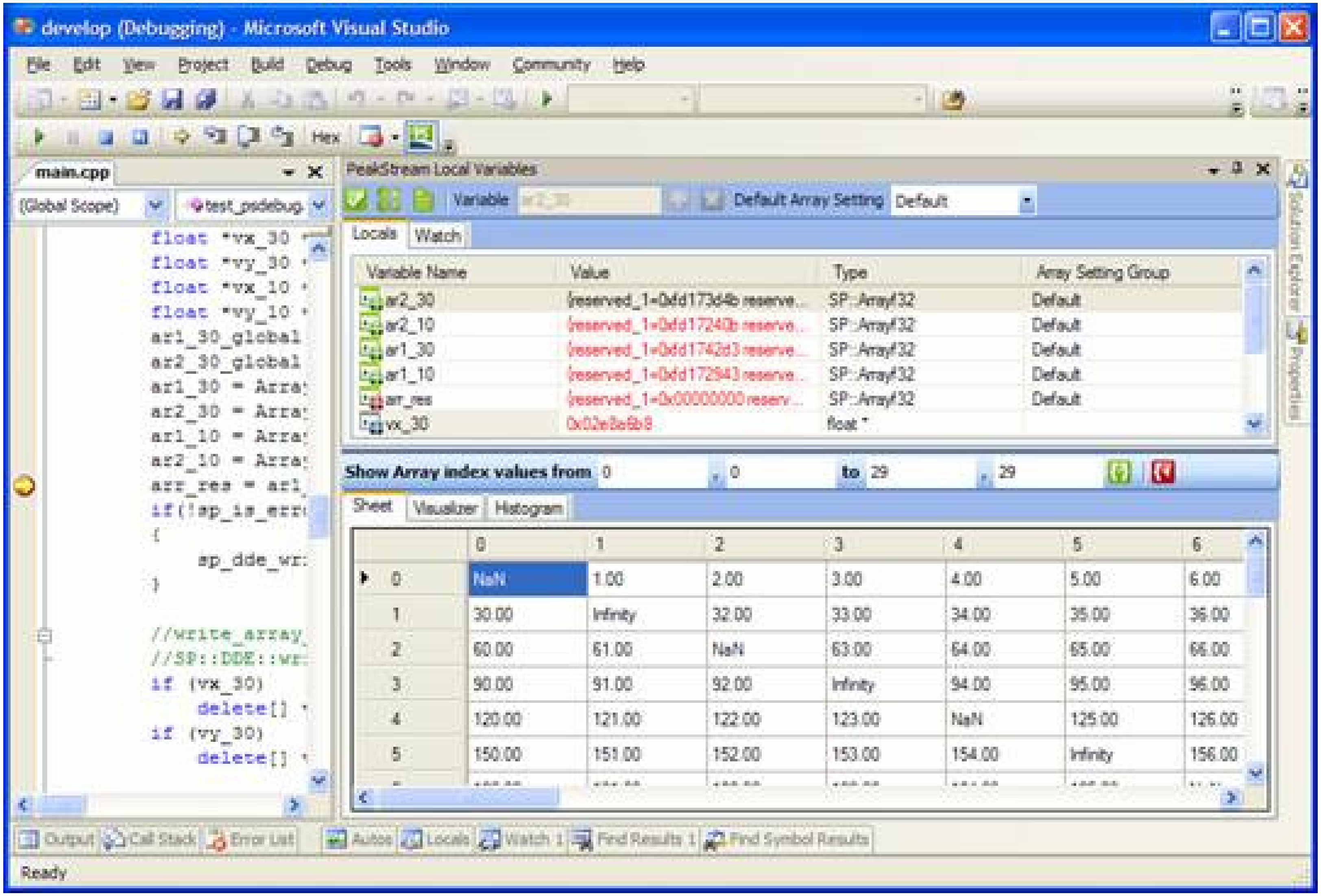
- **Role: report about what's inside a compute kernel**





PEAKSTREAM

# PeakStream Tools Extensions: Windows Debugger

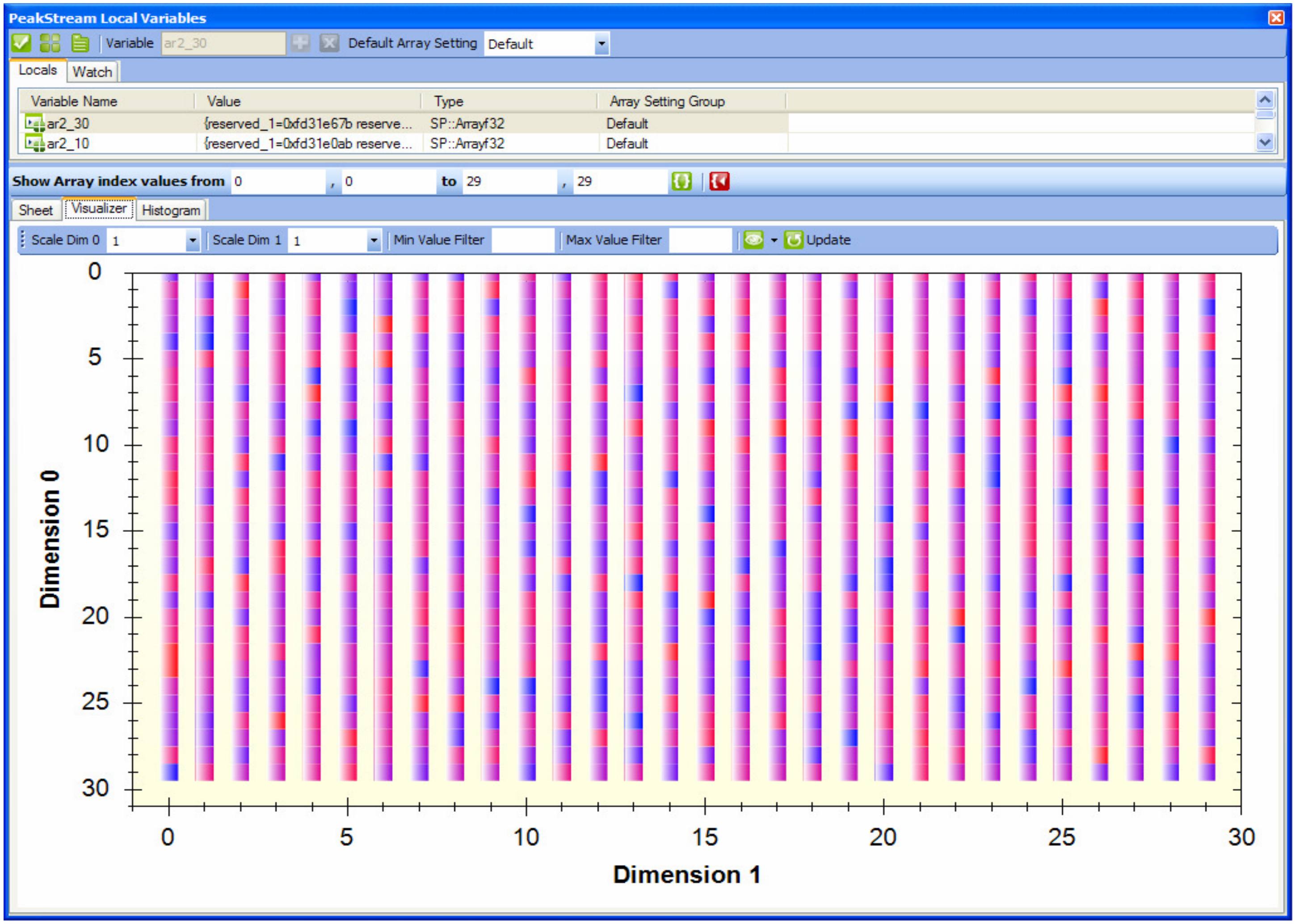







PEAKSTREAM

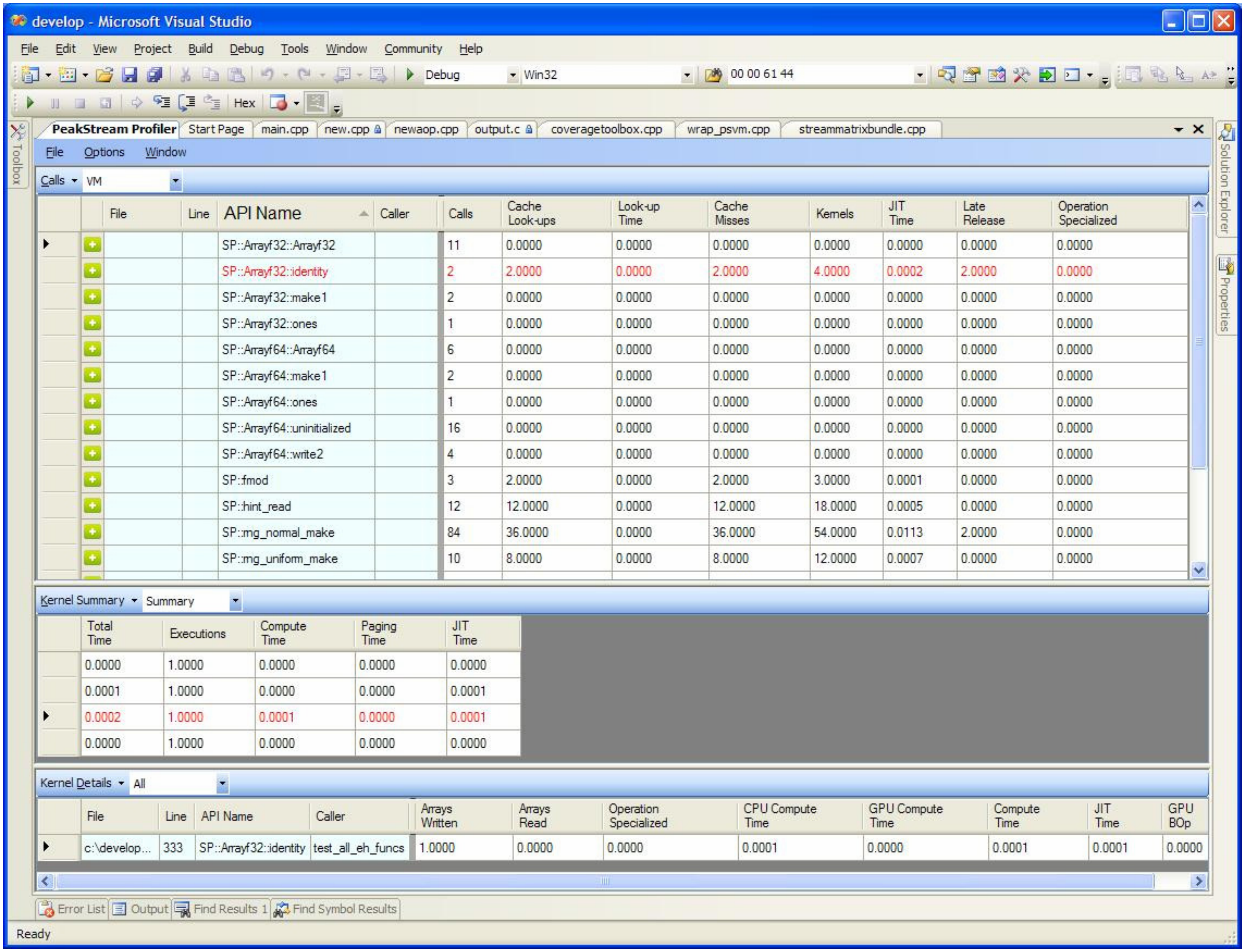
# PeakStream Tools Extensions: Windows Visualizer






PEAKSTREAM

# PeakStream Tools Extensions: Windows Profiler



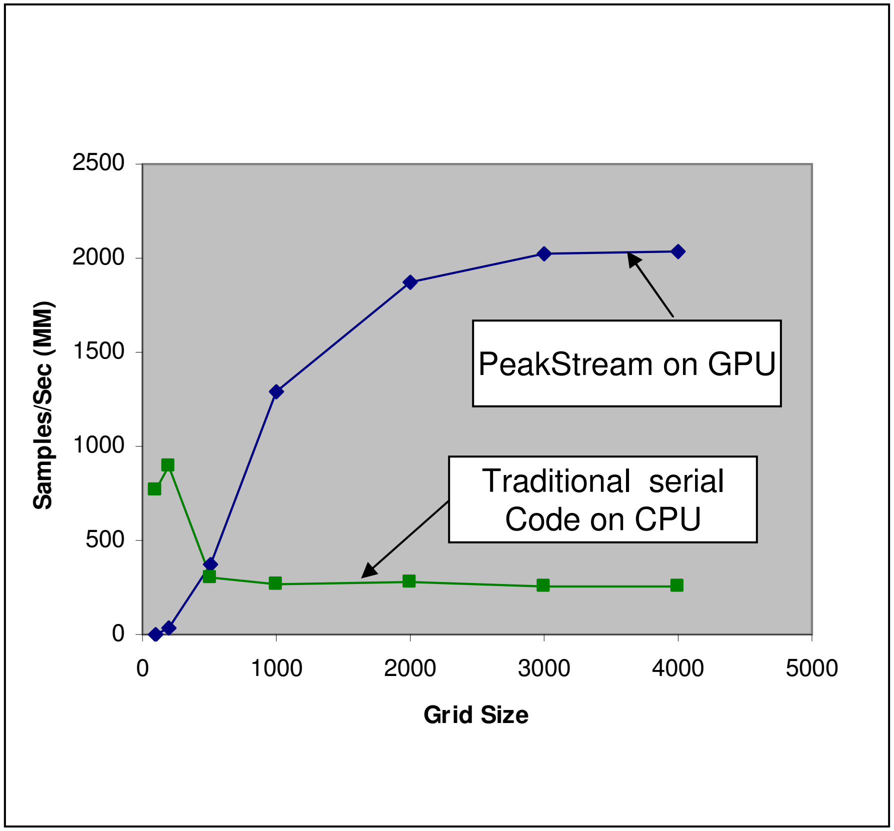




PEAKSTREAM

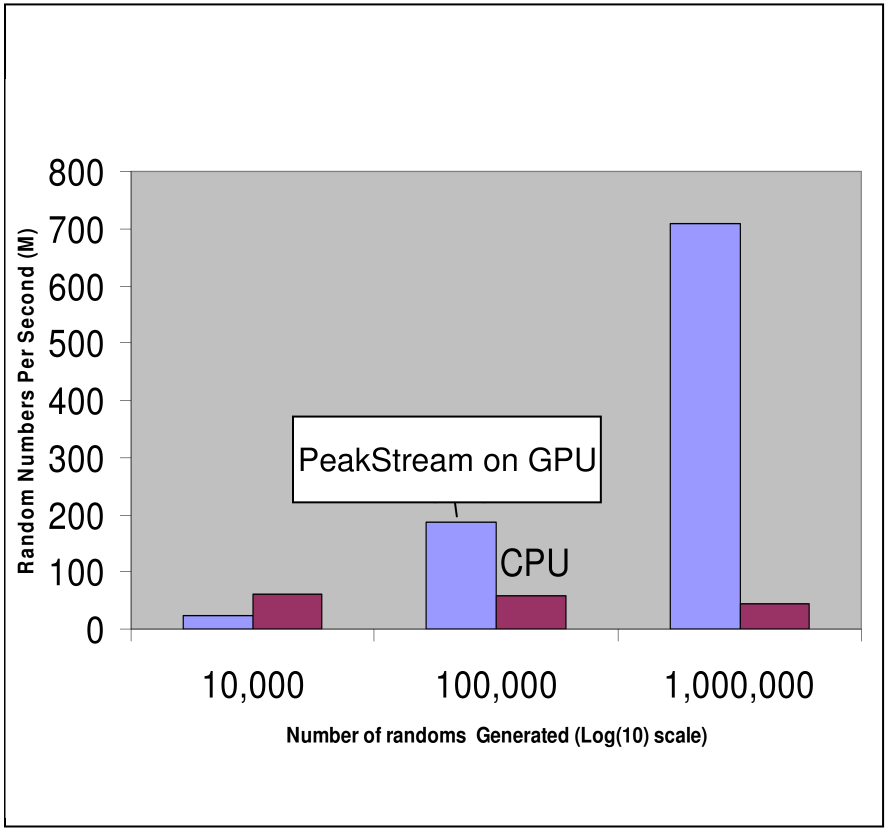
# Lab Application Benchmarks

Oil & Gas: Kirchhoff Migration




8x Peak Performance Advantage

Finance: Monte Carlo Simulation




16x Peak Performance Advantage



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;
}
```

# 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 ;
}
```





PEAKSTREAM

# The Future of Processors

» **Where are Processors going?**

» **Integrated CPU + GPU**
- AMD’s Fusion
- Intel’s Larrabee

» **GPUs and Cell Processor are not so different**
- They will converge
- And become integrated with a few CPU cores
- What differentiates them from CPUs?
  Explicit communication models

» **The Future Processor**
- A control processor (a.k.a. CPU)
- A compute array (GPU, Cell, etc.)

» **This is an excellent processor for both gaming and HPC**

# Where is Software for Many-Core Processors Going?

» **New programming models**
- Data-parallel is one approach, and there are others
- But the manually threaded approach leaves a lot to be desired
- How do we expose new models? APIs & languages

» **Increasing importance of runtime systems**
- Managing the processors: Scheduling
- Managing the data: Memory Management
- Managing the code: JITing

» **Reliance on compilers**
- To create optimal compute kernels for rapidly evolving processors
- In a way that protects the investment in application codes



PEAKSTREAM

# Conclusion

» **The world needs good programming environments to make parallel programming easier**
  - This is an exciting area of continued research
  - The need will persist for a long time

» **PeakStream was one such solution**
  - A data-parallel model for programming many-core
  - What other solutions can you think of?

» **Thank you very much**



PEAKSTREAM