Quantifi

Developing CUDA Accelerated .NET Plugins for Excel

NVIDIA 2010 Conference

Quantifi

Cuda Development

XLDeveloper-Cuda enabled

Cuda in a larger organization/codebase

Quantifi

XLDeveloper: Motivation

- Provide productive environment for quants to develop Excel user-defined functions
- Take care of the details of memory management, data marshaling, and exception handling
- Provide tools for drilling down into complex data structures such as curves and surfaces
- Support side-by-side deployment of multiple versions of Quantifi XL

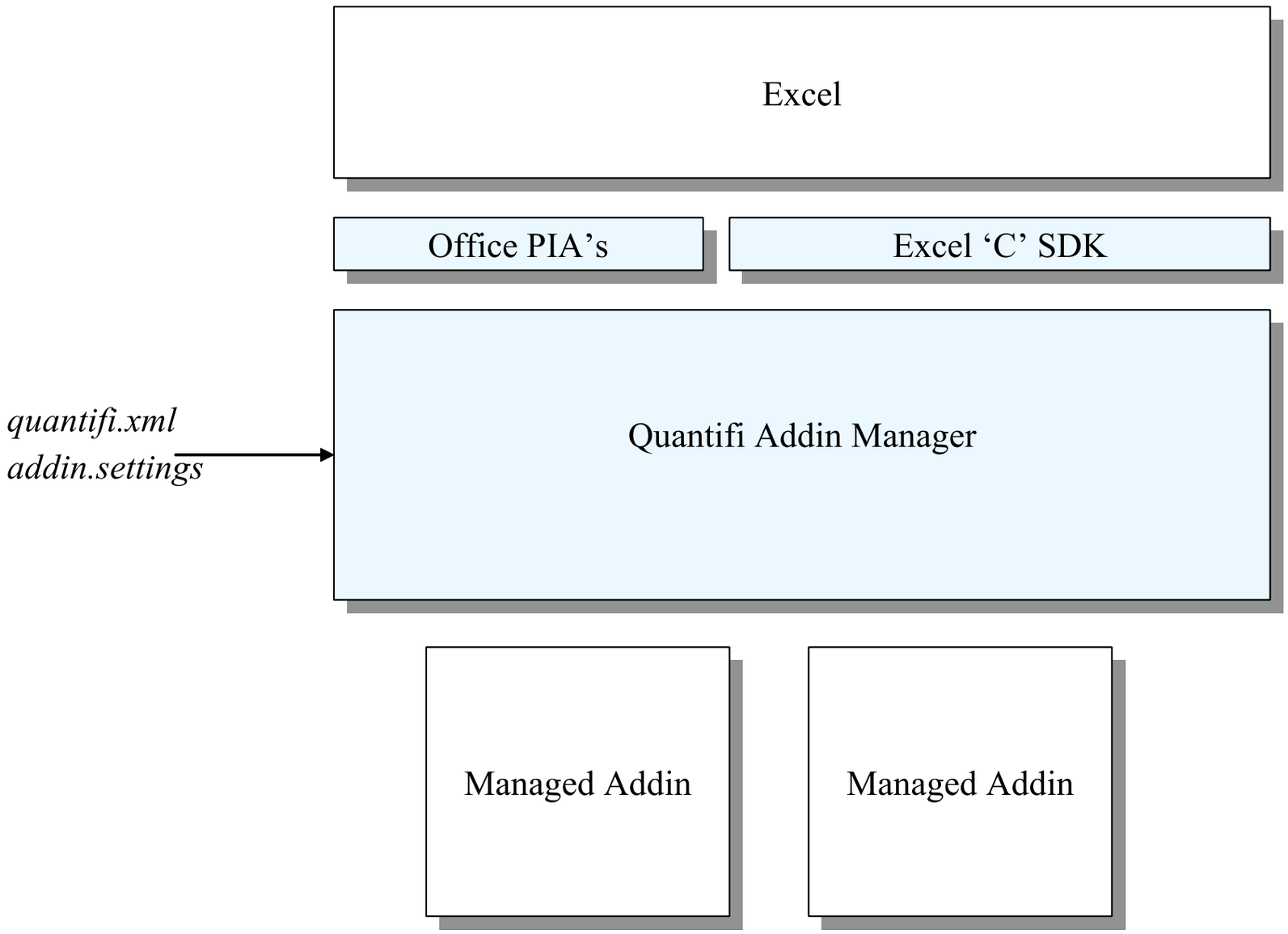
Quantifi

History

- 2003 - Original version developed
- 2005 - Major iteration to apply lessons learned and take advantage of .NET 2.0
- 2008 – Add support for Excel 2007 Ribbon bar

Quantifi

Architecture Overview



Quantifi

Supported Languages/Platforms

- OS
  - Windows XP/Vista/7
  - Windows Server 2003/2008
- Excel
  - 2002/2003/2007/2010
- Programming Languages
  - Any .NET language that supports custom attributes
    - e.g. C#, VB.NET, F#, C++/CLI
  - Easy to call C/C++

Quantifi

Key Features

- Simple Declarative Markup
  - Automated memory management
  - Automated data marshaling / validation
  - Automated generation of docs
- Object browsing
- Extensible Excel menu/ribbon bar
- Optional add on modules provides seamless integration with CUDA and Grid frameworks

Quantifi

Simple Declarative Markup

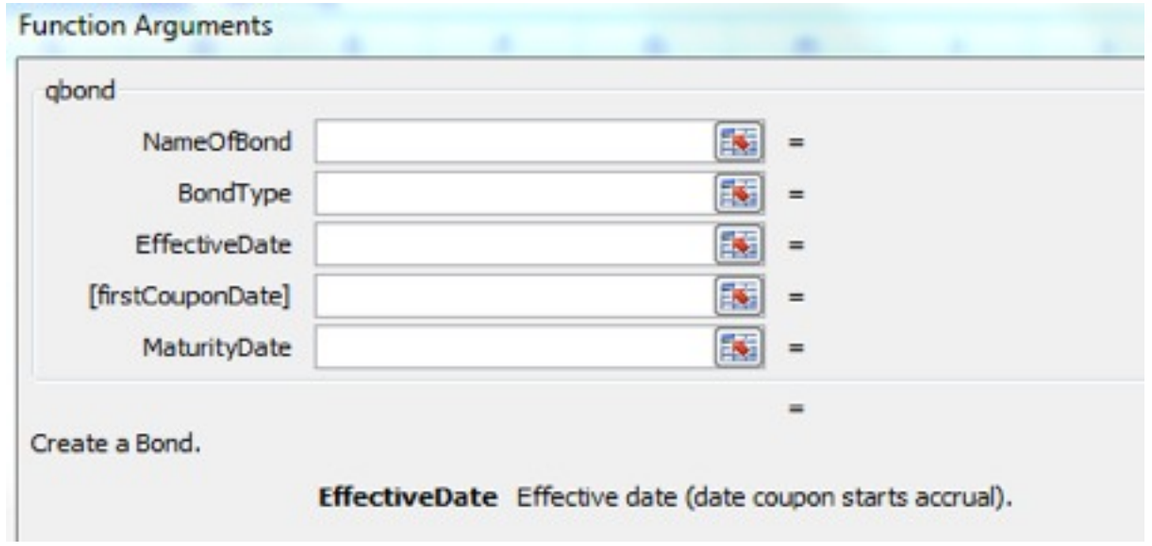
```
/// <summary>
/// simple addin class
/// </summary>
[XllClass("SimpleAddin")]
public class SimpleAddin
{
  /// <summary>
  /// Return the average of the specified array of doubles
  /// </summary>
  /// <param name="data">An array of doubles</param>
  /// <returns></returns>
  [XllFunction("Reduce")]
  public double Reduce(double[] data)
  {
    double total = 0.0;
    for( int i = 0; i < data.Length; i++ )
      total += data[i];

    return total/data.Length;
  }
}
```

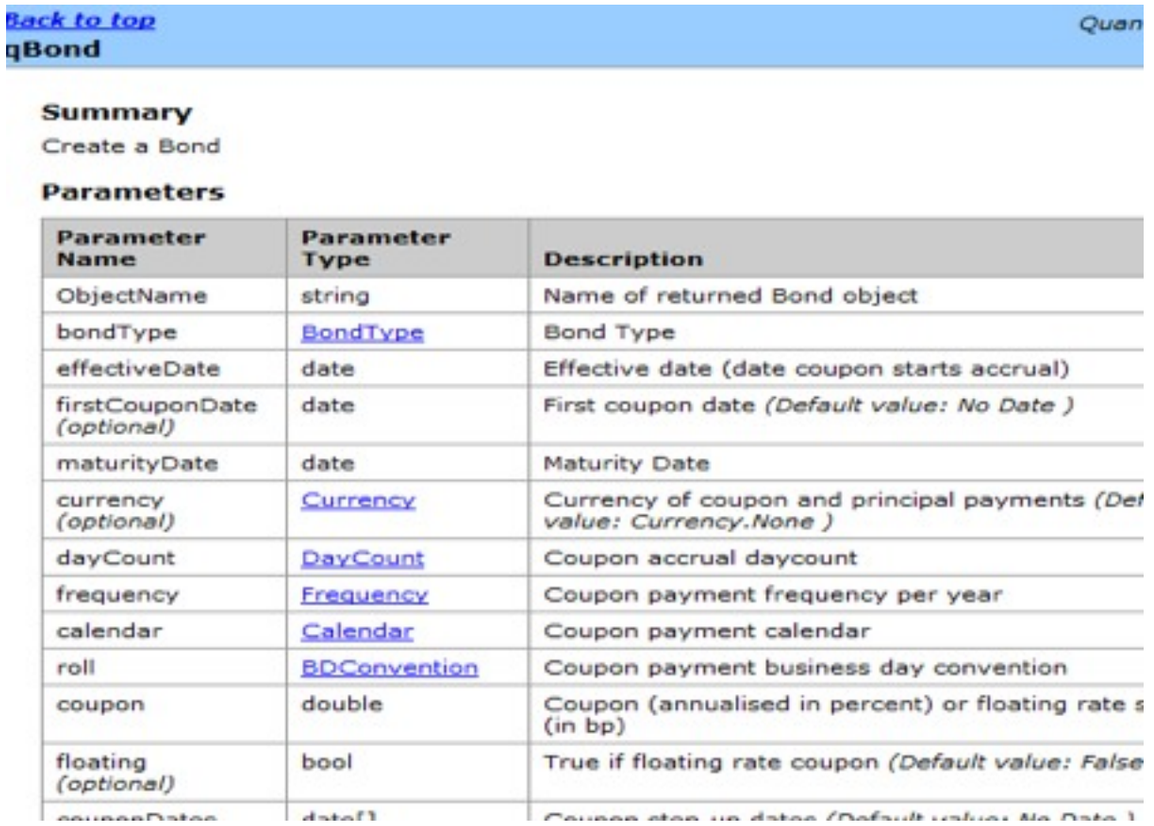
Quantifi

Automated Generation of Docs

- Function Wizard Help
  - Brief help for each arguments



- Detailed Function Help
  - Generated from code comments
  - Supports embedded Latex formulas



Quantifi

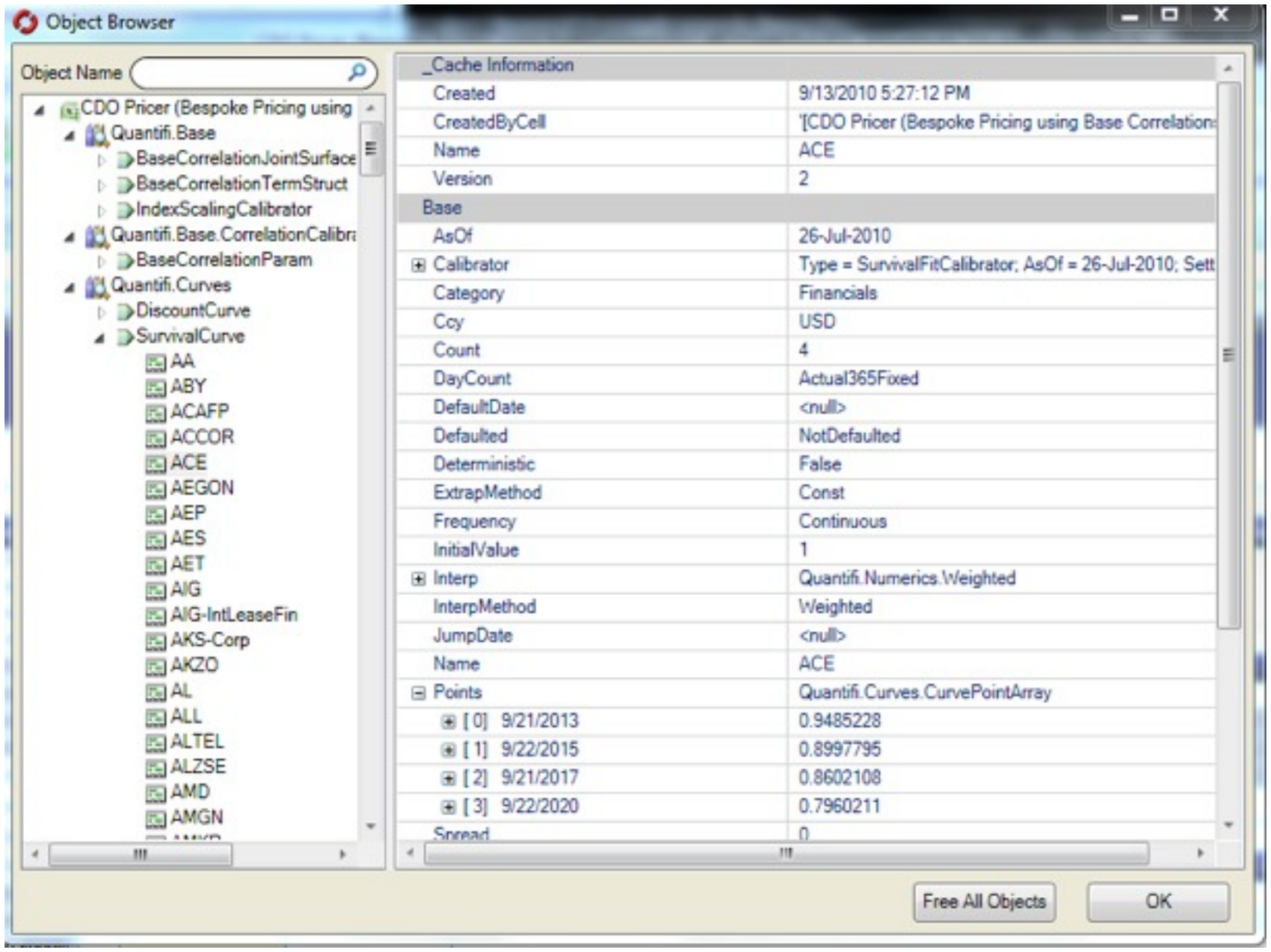
Extensive Datatype Support

Type	Excel	.NET
Double	NUM	Double
Char	Character	Char
Integer	INTEGER	Int32, Int64
String	STRING	String
Date	DATE	DateTime, Double
Enum	STRING	Enum
DataTable	RANGE	DataTable
Object	STRING	User Defined Type
Arrays	ARRAY, RANGE	Array

Quantifi

Object Browsing

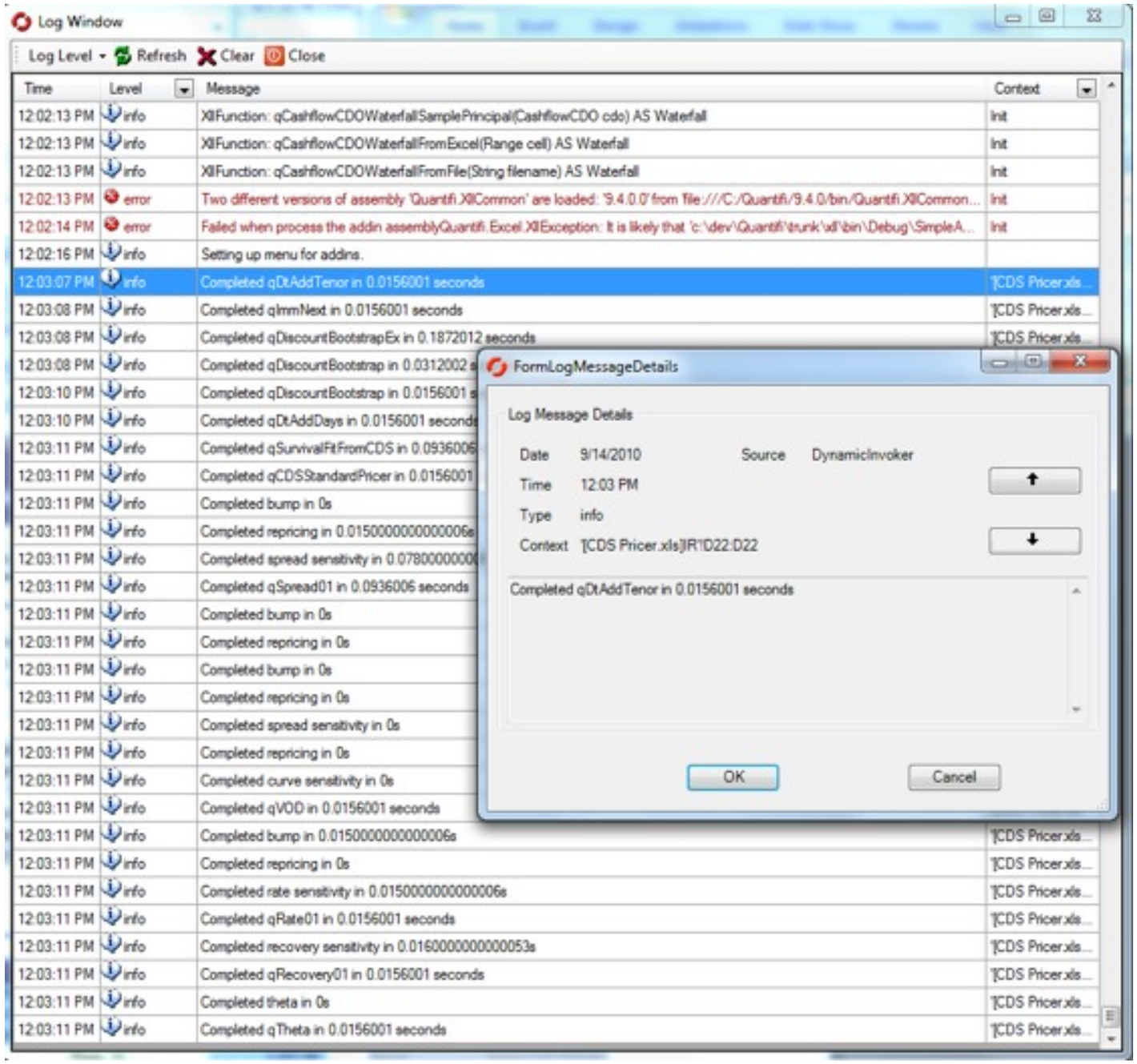
- Integrated Object Browser allows user to inspect any cached object
- Can view value of any public property



Quantifi

Log Message Window

- Addin can log message at different levels
- Menu Configurable by the user
- The context information (i.e. calculating Cell) is auto captured.

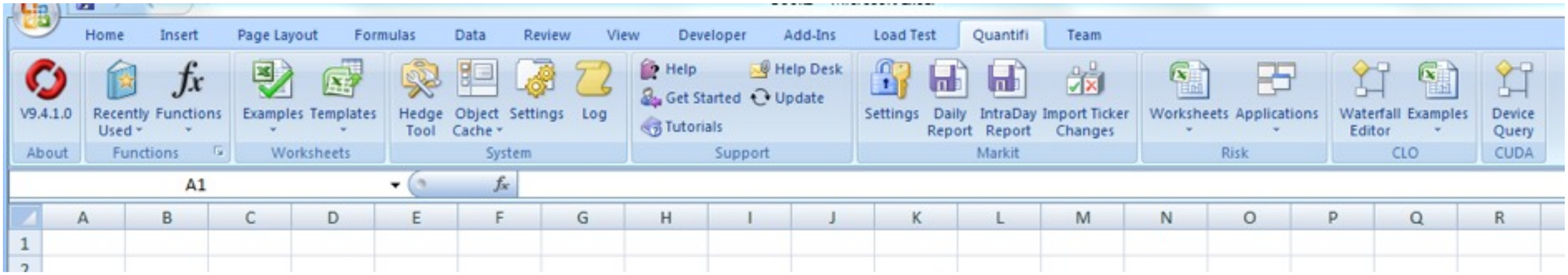


Quantifi

Extensible Excel Menu/Ribbon Bar

- Extension points provided to allow for easy addition of custom menus

- In addition to user-defined functions, entire applications can be built



- If using Excel 2007, menus are displayed in Ribbon Bar

Quantifi

Hybrid CPU/GPU Development

- The challenge:

  Creating Maintainable Code

  Allow for Specialized Development

  Allow for CPU and GPU Hardware

  Compilation with or without Cuda Hardware

- The tools:

  Templates

  Classes (Fermi)

Quantifi

An example

- Lots of modeling is around Normal Distributions;
- For example Uniformly (0,1) numbers need to be mapped to a normal distribution;
- Function below is not invertible
- So polynomial approximations...
- Monte Carlo example has one million paths, 40 time slices

Cumulative distribution function [edit]

The cumulative distribution function (cdf) describes probabilities for a random variable to fall in the intervals of the form $(-\infty, x]$. The cdf of the standard normal distribution is denoted with the capital Greek letter Φ (phi), and can be computed as an integral of the probability density function:

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2}\,dt = \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right], \quad x \in \mathbb{R}.$$

This integral can only be expressed in terms of a special function **erf**, called the error function. The numerical methods for calculation of the standard normal cdf are discussed below. For a generic normal random variable with mean μ and variance $\sigma^2 > 0$ the cdf will be equal to

$$F(x;\,\mu,\sigma^2) = \Phi\left(\frac{x-\mu}{\sigma}\right) = \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right], \quad x \in \mathbb{R}.$$

For a normal distribution with zero variance, the cdf is the Heaviside step function:

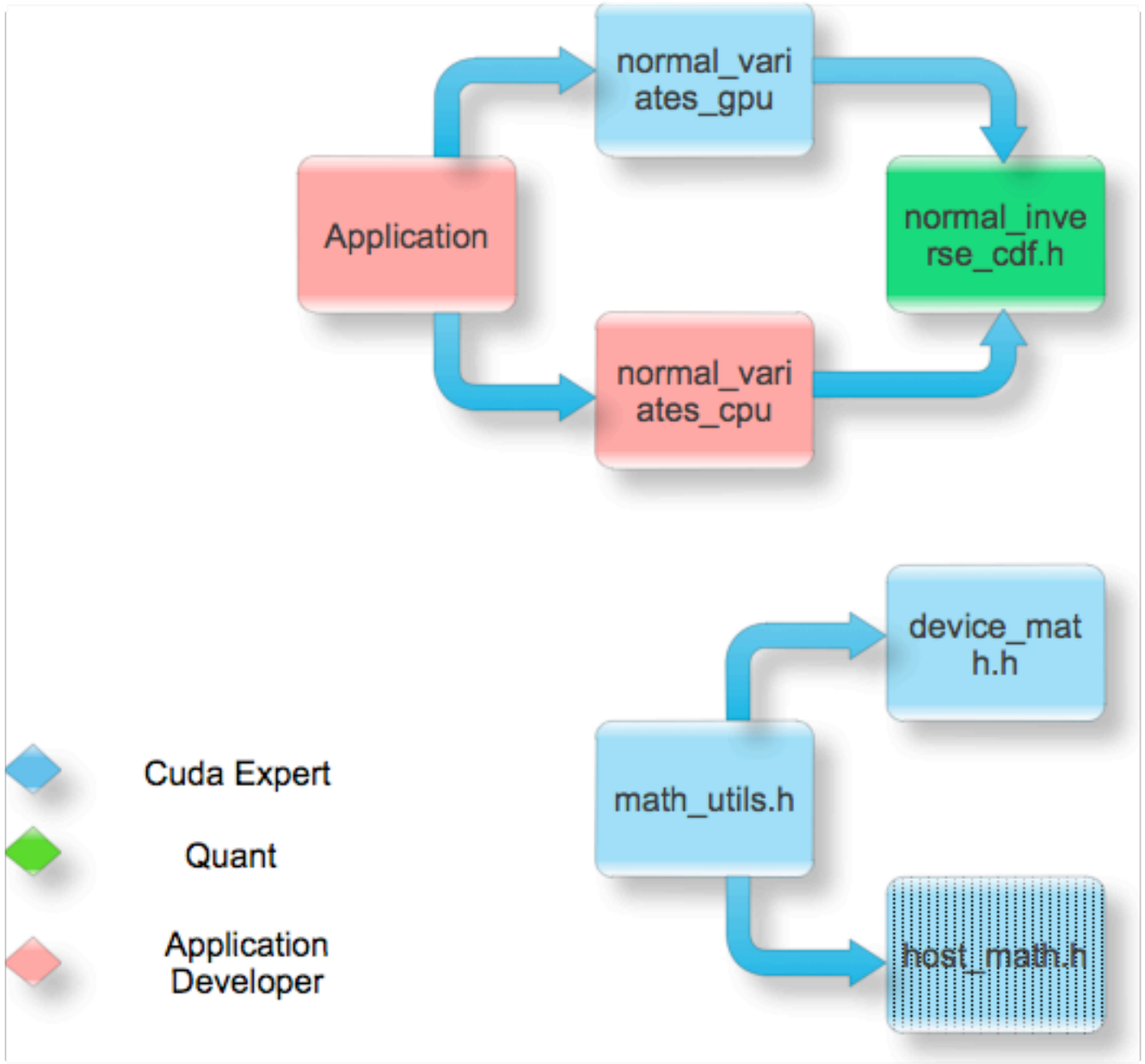
$$F(x;\,\mu,0) = \mathbf{1}\{x \geq \mu\}\,.$$

The complement of the standard normal cdf, $Q(x) = 1 - \Phi(x)$, is referred to as the Q-function, especially in engineering texts.[6][7] This represents the tail probability of the Gaussian distribution, that is the probability that a standard normal random variable X is greater than the number x. Other definitions of the Q-function, all of which are simple transformations of Φ, are also used occasionally.[8]

Source Wikipedia

Quantifi

CPU/GPU Development



Quantifi

Math_utils.h

- Mapping of functions
- Separate implementations on CPU and GPU
- In single precision and double precision
- Maintained by hardware expert
- device_math.h is CUDA implementation
- host_math.h is CPU implementation

math_utils.h

```
#ifdef __CUDA_ARCH__
#  include "details/device_math.h"
#else //!__CUDA_ARCH__
#  include "details/host_math.h"
#endif //__CUDA_ARCH__
```

GPU-device_math.h

```
/*
 * device_math.h
 *
 * Mathematical functions available on the device.
 *
 * Copyright (c) Quantifi Inc 2002-2011. All rights reserved.
 *
 */
#ifndef QN_DEVICE_MATH_H
#define QN_DEVICE_MATH_H

#include <host_defines.h>
#include <math_constants.h>

namespace qn { namespace cuda { namespace math {

  template<typename T> struct math_limits { };

template<> struct math_limits<float>
{
  __device__ inline static float infinity() { return CUDART_INF_F; }
  __device__ inline static float quiet_NaN() { return CUDART_NAN_F; }
  __device__ inline static float denorm_min() { return CUDART_MIN_DENORM_F; }
};

template<> struct math_limits<double>
{
  __device__ inline static double infinity() { return CUDART_INF; }
  __device__ inline static double quiet_NaN() { return CUDART_NAN; }
  __device__ inline static double denorm_min() { return CUDART_MIN_DENORM; }
};
```


Quantifi

GPU continued

```
template<class T> __device__ inline T fast_pow(T x, T y) {
  return pow(x,y);
}
template<class T> __device__ inline T fast_exp(T x) {
  return exp(x);
}
template<class T> __device__ inline T fast_log(T x) {
  return log(x);
}
template<class T> __device__ inline T fast_sin(T x) {
  return sin(x);
}
template<class T> __device__ inline T fast_cos(T x) {
  return cos(x);
}

__device__ inline float fast_pow(float x, float y) {
  return __powf(x,y);
}
__device__ inline float fast_exp(float x) {
  return __expf(x);
}
__device__ inline float fast_log(float x) {
  return __logf(x);
}
__device__ inline float fast_sin(float x) {
  return __sinf(x);
}
__device__ inline float fast_cos(float x) {
  return __cosf(x);
}
```


Quantifi

Cpu version: host_math.h

```
/*
 * host_math.h
 *
 * Mathematical functions available on the host.
 *
 * Copyright (c) Quantifi Inc 2002-2011. All rights reserved.
 *
 */
#ifndef QN_HOST_MATH_H
#define QN_HOST_MATH_H

#include <cmath>
#include <limits>
#include <algorithm>

namespace qn { namespace cuda { namespace math {

  template<typename T> struct math_limits : public std::numeric_limits<T>{};

  using std::abs;
  using std::fabs;
  using std::ceil;
  using std::floor;
  using std::sqrt;
  using std::pow;
  using std::log;
  using std::log10;
  using std::fmod;
  using std::modf;
  using std::exp;
  using std::frexp;
  using std::ldexp;
  using std::asin;
  using std::sin;
  using std::sinh;
```


Quantifi

Cpu version: host_math.h

```
using std::tanh;

using std::max;

using std::min;

static inline unsigned MUL(unsigned a, unsigned b){
  return a*b;
}

// overloads to match the device definitions
template<class T> inline T fast_pow(T x, T y) {
  return pow(x,y);
}
template<class T> inline T fast_exp(T x) {
  return exp(x);
}
  }
  template<class T> inline T fast_log(T x) {
    return log(x);
  }
  template<class T> inline T fast_sin(T x) {
    return sin(x);
  }
  template<class T> inline T fast_cos(T x) {
    return cos(x);
  }

}}} // namespace qn::cuda::details

#endif // QN_HOST_MATH_H
```


Quantifi

normal_inverse_cdf.h

- The actual inverse normal calculation
- Maintained by analytical expert
- Standard c++ code
- CPU/GPU agnostic
- Can be debugged on CPU
- We will show 3 implementations:
  - ShawBrickman:
  - Moro:
  - Acklam:

40 Million Calls(in millisec.) C2050

256x256	
ShawBrickman	9.97896
Moro	10.2861
Wichura	18.1781
Acklam	25.5947

128x128	
ShawBrickman	12.6027
Moro	13.3601
Wichura	18.1781
Acklam	30.7807

512x512	
ShawBrickman	9.78561
Moro	10.0949
Wichura	17.8182
Acklam	24.8159

BGM: one million paths, 40 timeslices

As a 3.1 sidenote: erfinv float is really fast but double....

Quantifi

CPU (one core) versus GPU

CPU timing of normal_inverse_cdf:

ErfInv	2877.03
ShawBrickman	4381.32
Moro	2632.27
Wichura	3160.79
Acklam	2028.23

GPU Timing of normal_inverse_cdf:

ErfInv	4.5177
ShawBrickman	9.74522
Moro	10.6267
Wichura	19.2429
Acklam	25.4905

CPU Time/GPU Time

ErfInv	636.51
ShawBrickman	449.37
Moro	247.63
Wichura	164.29
Acklam	79.57

Quantifi

ShawBrickman

```
include <host_defines.h>
include “math_utils.h”

namespace qn { namespace cuda { namespace math {
namespace ShawBrickman {
    template <typename Real>
    __device__  inline Real normal_inverse_cdf(Real y)
    {
      const Real P1 = Real(1.25331368352120879);
      const Real P2 = Real(1.97971542232292671);
      const Real P3 = Real(0.80002295072483916762);
      const Real P4 = Real(0.08740324826595858062);
      const Real P5 = Real(0.00207514095537565729 17);
      const Real P6 = Real(4.74482073242797246 2e-6);
      const Real Q1 = Real(1.0);
      const Real Q2 = Real(2.07955843605345893 11);
      const Real Q3 = Real(1.24993281173416030 14);
      const Real Q4 = Real(0.23668431621373705623);
      const Real Q5 = Real(0.01200982705591977 68);
      const Real Q6 = Real(0.00010596206919921025259);
      Real z;
      int sgn = (y >= Real(0.5));
      sgn = sgn - !sgn;
      z = -log(Real(1.0) - (sgn * ((Real(2.0) * y) - Real(1.0))));
      return sgn * z * (P1+z*(P2+z*(P3+z*(P4+(P5+P6*z)*z))))
        / (Q1+z*(Q2+z*(Q3+z*(Q4+(Q5+Q6*z)*z))));
    }
  } // namespace ShawBrickman

}}}
```


Quantifi

Moro

- namespace Moro {

```
////////////////////////////////////////////////////////////////////////////////
// Moro's Inverse Cumulative Normal Distribution function approximation
////////////////////////////////////////////////////////////////////////////////
template<class Real>
__device__ inline Real normal_inverse_cdf(Real P){
  using namespace qn::cuda::math;

  const Real a1 = Real(2.50662823884);
  const Real a2 = Real(-18.61500062529);
  const Real a3 = Real(41.39119773534);
  const Real a4 = Real(-25.44106049637);
  const Real b1 = Real(-8.4735109309);
  const Real b2 = Real(23.08336743743);
  const Real b3 = Real(-21.06224101826);
  const Real b4 = Real(3.13082909833);
  const Real c1 = Real(0.337475482272615);
  const Real c2 = Real(0.976169019091719);
  const Real c3 = Real(0.160797971491821);
  const Real c4 = Real(2.76438810333863E-02);
  const Real c5 = Real(3.8405729373609E-03);
  const Real c6 = Real(3.951896511919E-04);
  const Real c7 = Real(3.21767881768E-05);
  const Real c8 = Real(2.888167364E-07);
  const Real c9 = Real(3.960315187E-07);

  if(P <= 0 || P >= Real(1.0))
    return F::invalid_value();

  Real y, z;
  y = P - Real(0.5);
  if(fabsf(y) < Real(0.42)){
    z = y * y;
    z = y * (((a4 * z + a3) * z + a2) * z + a1) / ((((b4 * z + b3) * z + b2) * z + b1) * z + Real(1.0));
  }else{
    if(y > 0)
      z = fast_log(-fast_log(Real(1.0) - P));
    else
      z = fast_log(-fast_log(P));

    z = c1 + z * (c2 + z * (c3 + z * (c4 + z * (c5 + z * (c6 + z * (c7 + z * (c8 + z * c9)))))));
    if(y < 0) z = -z;
  }

  return z;
```


Quantifi

Acklam

- namespace Acklam {

```
////////////////////////////////////////////////////////////////////////////////
// Acklam's Inverse Cumulative Normal Distribution function approximation
////////////////////////////////////////////////////////////////////////////////
template<class Real>
__device__ inline Real normal_inverse_cdf(Real P){
  using namespace qn::cuda::math;

  const Real   a1 = Real(-39.6968302866538);
  const Real   a2 = Real(220.946098424521);
  const Real   a3 = Real(-275.928510446969);
  const Real   a4 = Real(138.357751867269);
  const Real   a5 = Real(-30.6647980661472);
  const Real   a6 = Real(2.50662827745924);
  const Real   b1 = Real(-54.4760987982241);
  const Real   b2 = Real(161.585836858041);
  const Real   b3 = Real(-155.698979859887);
  const Real   b4 = Real(66.8013118877197);
  const Real   b5 = Real(-13.2806815528857);
  const Real   c1 = Real(-7.78489400243029E-03);
  const Real   c2 = Real(-0.322396458041136);
  const Real   c3 = Real(-2.40075827716184);
  const Real   c4 = Real(-2.54973253934373);
  const Real   c5 = Real(4.37466414146497);
  const Real   c6 = Real(2.93816398269878);
  const Real   d1 = Real(7.78469570904146E-03);
  const Real   d2 = Real(0.32246712907004);
  const Real   d3 = Real(2.445134137143);
  const Real   d4 = Real(3.75440866190742);
  const Real  low = Real(0.02425);
  const Real high = Real(1.0) - low;
  Real z, R;

  if(P <= 0 || P >= 1.0f)
    return math_limits<Real>::quiet.NaN();

  if(P < low){
    z = sqrt(Real(-2.0) * log(P));
    z = (((((c1 * z + c2) * z + c3) * z + c4) * z + c5) * z + c6) /
      ((((d1 * z + d2) * z + d3) * z + d4) * z + Real(1.0));
  }else{
    if(P > high){
      z = sqrt(-2.0 * log(1.0 - P));
      z = -(((((c1 * z + c2) * z + c3) * z + c4) * z + c5) * z + c6) /
        ((((d1 * z + d2) * z + d3) * z + d4) * z + Real(1.0));
    }else{
      z = P - Real(0.5);
      R = z * z;
      z = (((((a1 * R + a2) * R + a3) * R + a4) * R + a5) * R + a6) * z /
        (((((b1 * R + b2) * R + b3) * R + b4) * R + b5) * R + Real(1.0));
    }
  }
```

normal_variate_cpu.cpp

```
#include <cutil_inline.h>


#include "normal_inverse_cdf.h"

namespace qn { namespace cuda {
  void normal_inverse_cdf_cpu(float *h_Output, float *h_Input, unsigned int N)
  {
    using qn::cuda::details::normal_inverse_cdf;
    for (unsigned int i = 0; i < N; ++i)
      h_Output[i] = normal_inverse_cdf(h_Input[i]);
  }
}} // namespace qn::cuda
```

normal_variate_gpu.cpp

- sets up the kernel calls
- has kernel which calls common code
- maintained by cuda expert

normal_variate_gpu

```
#include <cutil_inline.h>

#include "normal_inverse_cdf.h"

namespace qn { namespace cuda {namespace details {
  ////////////////////////////////////////////////////////////////////////////
  // Main kernel. Choose between transforming
  // input sequence and uniform ascending (0, 1) sequence
  ////////////////////////////////////////////////////////////////////////////
  template<class Real, class InvF>
  static __global__ void inverseCNDKernel(
    Real *d_Output, Real *d_Input, unsigned int pathN, InvF inverse)
  {
    using qn::cuda::math::MUL;

    Real q = Real(1.0) / (Real)(pathN + 1);
    unsigned int     tid = MUL(blockDim.x, blockIdx.x) + threadIdx.x;
    unsigned int threadN = MUL(blockDim.x, gridDim.x);

    //Transform input number sequence if it's supplied
    if(d_Input){
      for(unsigned int pos = tid; pos < pathN; pos += threadN){
        Real d = d_Input[pos];
        d_Output[pos] = (Real)inverse(d);
      }
    }
    //Else generate input uniformly placed samples on the fly
    //and write to destination
    else{
      for(unsigned int pos = tid; pos < pathN; pos += threadN){
        Real d = (Real)(pos + 1) * q;
        d_Output[pos] = (Real)inverse(d);
      }
    }
  }
}}} // namespace qn::cuda::details
```


Quantifi

normal_variate_gpu

```
namespace qn { namespace cuda {
namespace details {
template<typename Real>
struct Moro {
  __device__ inline Real operator()(Real x) const {
    return qn::cuda::math::Moro::normal_inverse_cdf(x);
  }
};

template<typename Real>
struct ShawBrickman {
  __device__ inline Real operator()(Real x) const {
    return qn::cuda::math::ShawBrickman::normal_inverse_cdf(x);
  }
};

template<typename Real>
struct Acklam {
  __device__ inline Real operator()(Real x) const {
    return qn::cuda::math::Acklam::normal_inverse_cdf(x);
  }
};
  }
};
}

const int NBLOCK = 512;

void normal_inverse_cdf_gpu(float *d_Output, float *d_Input, unsigned int N)
{
  qn::cuda::details::inverseCNDKernel<<<NBLOCK, NBLOCK>>>(
    d_Output, d_Input, N, details::ShawBrickman<float>());
  cutilCheckMsg("inverseCNDKernel() execution failed.\n");
}

void normal_inverse_cdf_gpu_Moro(float *d_Output, float *d_Input, unsigned int N)
{
  qn::cuda::details::inverseCNDKernel<<<NBLOCK, NBLOCK>>>(
    d_Output, d_Input, N, details::Moro<float>());
  cutilCheckMsg("inverseCNDKernel() execution failed.\n");
}

void normal_inverse_cdf_gpu_Acklam(float *d_Output, float *d_Input, unsigned int N)
{
  qn::cuda::details::inverseCNDKernel<<<NBLOCK, NBLOCK>>>(
    d_Output, d_Input, N, details::Acklam<float>());
  cutilCheckMsg("inverseCNDKernel() execution failed.\n");
}
}} // namespace qn::cuda
```


Quantifi

In conclusion

- With the right development tools, cuda can be easily integrated in a production development environment;
- Cuda has matured enough to co-exist with a large code base. Significant parts of development can be agnostic of the hardware platform it will be deployed on;
- Gpu development can organizationally co-exist within an organization;

Contact Details

Europe

Quantifi Limited
16 Martin's Le Grand
London, EC1A 4EN
UK
Ph: +44 20 7397 8788

North America

Quantifi Inc.
230 Park Avenue
New York, NY 10169
USA
Ph: +1 212 784 6815

Asia Pacific

Quantifi Pty Ltd
111 Elizabeth St
Sydney, NSW 2000
Australia
Ph: +61 2 9221 0133

Web: www.quantifisolutions.com

Email: enquire@quantifisolutions.com

Quantifi