CUDA Without Cuda (CUDA Libraries)

GPU Computing Webinar 7/16/2011

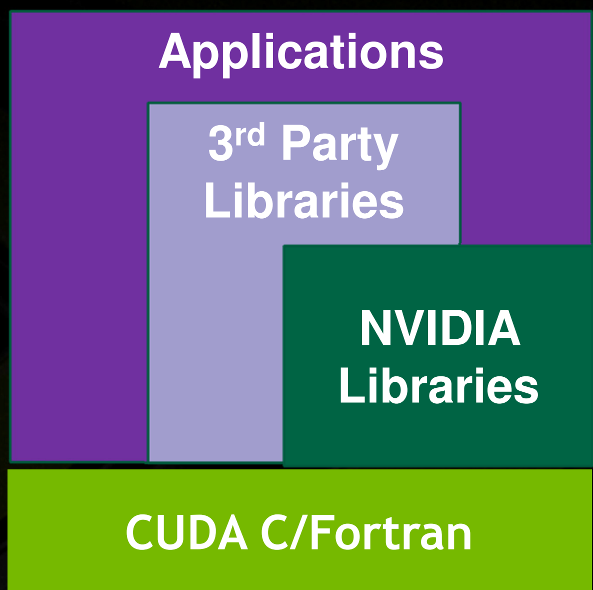
Dr. Justin Luitjens, NVIDIA Corporation

NVIDIA®

CUDA Accelerated Libraries

- Advantages of using these libraries
  - Already parallelized
  - Already implemented
  - Already debugged
  - Already optimized
- Disadvantages of using these libraries
  - ...
- NVIDIA provides a handful of libraries and there are also a number of 3rd party libraries already in existence.

NVIDIA CUDA Libraries



- CUFFT
- CUBLAS
- CUSPARSE
- Libm (math.h)
- CURAND
- NPP
- Thrust

Documentation for all NVIDIA libraries:

http://developer.nvidia.com/nvidia-gpu-computing-documentation

NVIDIA CUDA Libraries

CUDA Toolkit includes several libraries:

- CUFFT: Fourier transforms
- CUBLAS: Dense Linear Algebra
- CUSPARSE : Sparse Linear Algebra
- LIBM: Standard C Math library
- CURAND: Pseudo-random and Quasi-random numbers
- NPP: Image and Signal Processing
- Thrust : STL-Like Primitives Library

Several open source and commercial* libraries:

- MAGMA: Linear Algebra
- CULA Tools*: Linear Algebra
- CUSP: Sparse Linear Solvers
- CUDPP: Parallel Primitives Library
- And more…

- OpenVidia: Computer Vision
- OpenCurrent: CFD
- NAG*: Computational Finance

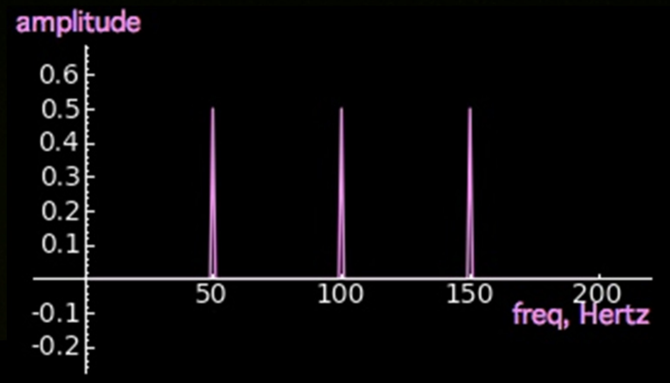
Disclaimer

- I am not an expert on all of these libraries
  - In fact, many of these libraries I have never used

- Today's Goals
  - Provide an overview of various libraries and their features but not necessarily how to use them.
  - For details on how to use them I suggest reading the libraries documentation.

CUFFT Library

CUFFT is a GPU based Fast Fourier Transform library



$$F(x) = \sum_{n=0}^{N-1} f(n) e^{-j2\pi(x\frac{n}{N})}$$

$$f(n) = \frac{1}{N} \sum_{n=0}^{N-1} F(x) e^{j2\pi(x\frac{n}{N})}$$

CUFFT Library Features

- Algorithms based on Cooley-Tukey ($n = 2^a \cdot 3^b \cdot 5^c \cdot 7^d$) and Bluestein
- Simple interface similar to FFTW
- 1D, 2D and 3D transforms of complex and real data
- Row-major order (C-order) for 2D and 3D data
- Single precision (SP) and Double precision (DP) transforms
- In-place and out-of-place transforms
- 1D transform sizes up to 128 million elements
- Batch execution for doing multiple transforms
- Streamed asynchronous execution
- Non normalized output: IFFT(FFT(A))=len(A)*A

CUFFT in 4 easy steps

<u>Step 1</u> – Allocate space on GPU memory

<u>Step 2</u> – Create plan specifying transform configuration like the size and type (real, complex, 1D, 2D and so on).

<u>Step 3</u> – Execute the plan as many times as required, providing the pointer to the GPU data created in Step 1.

<u>Step 4</u> – Destroy plan, free GPU memory

Example CUFFT Program

```
#include <stdlib.h>
#include <stdio.h>
#include "cufft.h"

#define NX 256
#define NY 128

main()
{
  cufftHandle plan;
  cufftComplex *idata, *odata;
  int i;

  cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
  cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*NY);
  for( i=0; i<NX*NY; i++ ) {
        idata[i].x = (float)rand() / (float)RAND_MAX;
        idata[i].y = (float)rand() / (float)RAND_MAX;
  }

  /* Create a 2D FFT plan. */
  cufftPlan2d(&plan, NX, NY, CUFFT_C2C);

  /* Use the CUFFT plan to transform the signal out of place.
   * Note: idata != odata indicates an out of place
   * transformation to CUFFT at execution time. */
  cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);

  /* Inverse transform the signal in place */
  cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);

  /* Destroy the CUFFT plan. */
  cufftDestroy(plan);
  cudaFree(idata);
  cudaFree(odata);

  return 0;
}
```

CUBLAS Library

- Implementation of BLAS (Basic Linear Algebra Subprograms)
  - Self-contained at the API level
- Supports all the BLAS functions
  - Level1 (vector,vector): O(N)
    - AXPY : y = alpha.x + y
    - DOT : dot = x.y
  - Level 2(matrix,vector): $O(N^{2})$
    - Vector multiplication by a General Matrix : GEMV
    - Triangular solver : TRSV
  - Level3(matrix,matrix): $O(N^{3})$
    - General Matrix Multiplication : GEMM
    - Triangular Solver : TRSM
- Following BLAS convention, CUBLAS uses column-major storage

CUBLAS Features

- Support of 4 types :
  - Float, Double, Complex, Double Complex
  - Respective Prefixes : S, D, C, Z
- Contains 152 routines : S(37), D(37), C(41), Z(41)
- Function naming convention: cublas + BLAS name
- Example: cublasSGEMM
    - S: single precision (float)
    - GE: general
    - M: multiplication
    - M: matrix

Using CUBLAS

- Interface to CUBLAS library is in cublas.h
- Function naming convention
  - cublas + BLAS name
  - Eg., cublasSGEMM
- Error handling
  - CUBLAS core functions do not return error
    - CUBLAS provides function to retrieve last error recorded
  - CUBLAS helper functions do return error
- Helper functions:
  - Memory allocation, data transfer

Calling CUBLAS from C

```
#include <stdlib.h>
#include <stdio.h>
#include "cublas.h"

main()
{
    float *a, *b, *c;
    float *d_a, *d_b, *d_c;
    int lda, ldb, ldc;
    int i, j, n;
    struct timeval t1, t2, t3, t4;
    double dt1, dt2, flops;

    cublasInit();
    printf( " n      t1      t2    GF/s  GF/s\n" );

    for( n=512; n<5120; n+=512 ) {

        lda = ldb = ldc = 2*n;

        cudaMallocHost( (void**)&a, n*lda*sizeof(float) );
        cudaMallocHost( (void**)&b, n*ldb*sizeof(float) );
        cudaMallocHost( (void**)&c, n*ldc*sizeof(float) );

        for( j=0; j<n; j++ ) {
            for( i=0; i<n; i++ ) {
                a[i+j*lda] = (float)rand()/(float)RAND_MAX;
                b[i+j*ldb] = (float)rand()/(float)RAND_MAX;
                c[i+j*ldc] = (float)rand()/(float)RAND_MAX;
            }
        }

        cublasAlloc( n*lda, sizeof(float), (void **)&d_a );
        cublasAlloc( n*ldb, sizeof(float), (void **)&d_b );
        cublasAlloc( n*ldc, sizeof(float), (void **)&d_c );

        gettimeofday (&t1, NULL);
        cublasSetMatrix( n, n, sizeof(float), a, lda, d_a, lda );
        cublasSetMatrix( n, n, sizeof(float), b, ldb, d_b, ldb );
        gettimeofday (&t2, NULL);
        cublasSgemm( 'N', 'N', n, n, n, 1.0, d_a, lda, d_b, ldb, 0.0, d_c, ldc );
        cudaThreadSynchronize();
        gettimeofday (&t3, NULL);
        cublasGetMatrix( n, n, sizeof(float), d_c, ldc, c, ldc );
        gettimeofday (&t4, NULL);

        cublasFree( d_a );
        cublasFree( d_b );
        cublasFree( d_c );

        cudaFreeHost( a );
        cudaFreeHost( b );
        cudaFreeHost( c );

        tdiff1 = t4.tv_sec - t1.tv_sec + 1.0e-6 * (t4.tv_usec - t1.tv_usec);
        tdiff2 = t3.tv_sec - t2.tv_sec + 1.0e-6 * (t3.tv_usec - t2.tv_usec);
        flops = 2.0 * (double)n * (double)n * (double)n;
        printf( "%4d %8.5f %8.5f %5.0f %5.0f\n", n, dt1, dt2, 1.0e-9*flops/tdiff1, 1.0e-9*flops/tdiff2 );
    }

    cublasShutdown();
    return 0;
}
```

Calling CUBLAS from FORTRAN

- **Two interfaces:**
  - **Thunking**
    - Allows interfacing to existing applications without any changes
    - During each call, the wrappers allocate GPU memory, copy source data from CPU memory space to GPU memory space, call CUBLAS, and finally copy back the results to CPU memory space and deallocate the GPGPU memory
    - Intended for light testing due to call overhead
  - **Non-Thunking** (default)
    - Intended for production code
    - Substitute device pointers for vector and matrix arguments in all BLAS functions
    - Existing applications need to be modified slightly to allocate and deallocate data structures in GPGPU memory space (using CUBLAS_ALLOC and CUBLAS_FREE) and to copy data between GPU and CPU memory spaces (using CUBLAS_SET_VECTOR, CUBLAS_GET_VECTOR, CUBLAS_SET_MATRIX, and CUBLAS_GET_MATRIX)

SGEMM example (THUNKING)

```
program example_sgemm
! Define  3  single precision matrices A, B, C
real, dimension(:,:),allocatable:: A(:,:),B(:,:),C(:,:)
integer:: n=16
allocate (A(n,n),B(n,n),C(n,n))
! Initialize A, B and C
...
#ifdef CUBLAS
 ! Call SGEMM in CUBLAS library using THUNKING interface (library takes care of
 ! memory allocation on device and data movement)
 call cublas_SGEMM('n','n', n,n,n,1.,A,n,B,n,1.,C,n)
#else
! Call SGEMM in  host BLAS library
  call SGEMM ('n','n',m1,m1,m1,alpha,A,m1,B,m1,beta,C,m1)
#endif
print *,c(n,n)
end program example_sgemm
```

To use the host BLAS routine:

```
g95 –O3 code.f90 –L/usr/local/lib -lblas
```

To use the CUBLAS routine (fortran_thunking.c is included in the toolkit /usr/local/cuda/src):

```
nvcc -O3 -c  fortran_thunking.c
g95 -O3 -DCUBLAS code.f90 fortran_thunking .o -L/usr/local/cuda/lib64 -lcudart  -lcublas
```

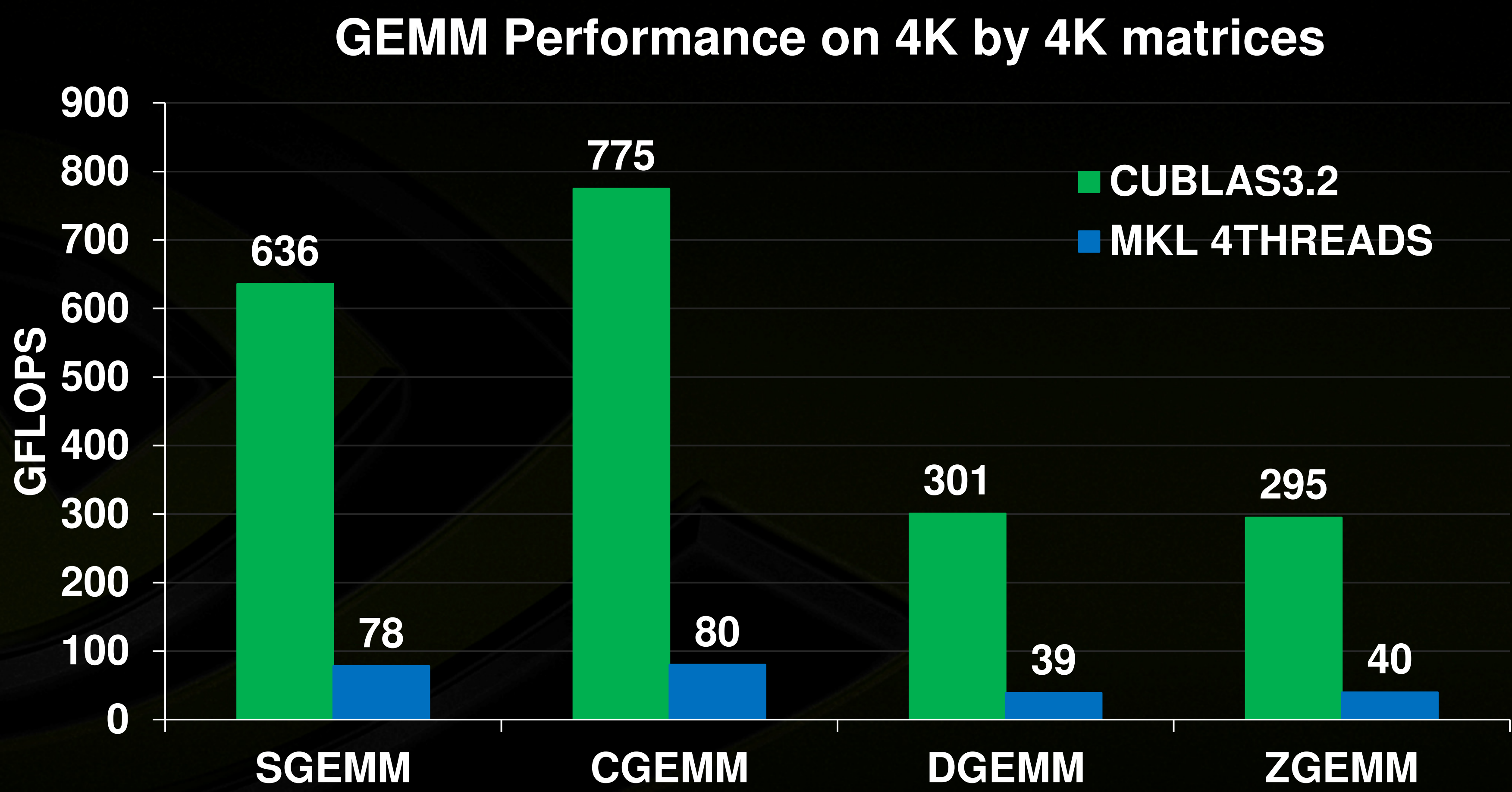
SGEMM example (NON-THUNKING)

```
program example_sgemm
real, dimension(:,:),allocatable:: A(:,:),B(:,:),C(:,:)
integer*8:: devPtrA, devPtrB, devPtrC
integer:: n=16, size_of_real=16
allocate (A(n,n),B(n,n),C(n,n))
call cublas_Alloc(n*n,size_of_real, devPtrA)
call cublas_Alloc(n*n,size_of_real, devPtrB)
call cublas_Alloc(n*n,size_of_real, devPtrC)
! Initialize A, B and C
...
! Copy data to GPU
call cublas_Set_Matrix(n,n,size_of_real,A,n,devPtrA,n)
call cublas_Set_Matrix(n,n,size_of_real,B,n,devPtrB,n)
call cublas_Set_Matrix(n,n,size_of_real,C,n,devPtrC,n)
! Call SGEMM in CUBLAS library
call cublas_SGEMM('n','n', n,n,n,1.,devPtrA,n,devPtrB,n,1.,devPtrC,n)
! Copy data from GPU
call cublas_Get_Matrix(n,n,size_of_real,devPtrC,n,C,n)
print *,c(n,n)
call cublas_Free(devPtrA)
call cublas_Free(devPtrB)
call cublas_Free(devPtrC)
end program example_sgemm
```

To use the CUBLAS routine (fortran.c is included in the toolkit /usr/local/cuda/src):

```
nvcc -O3 -c  fortran.c
g95 -O3  code.f90 fortran.o -L/usr/local/cuda/lib64 -lcudart  -lcublas
```

GEMM Performance



Performance may vary based on OS version and motherboard configuration

cuBLAS 3.2, Tesla C2050 (Fermi), ECC on
MKL 10.2.3, 4-core Corei7 @ 2.66 GHz

Changes to CUBLAS API in CUDA 4.0

- New header file cublas_v2.h: defines new API.
- Add a handle argument: gives the control necessary to manage multiple host threads and multiple GPUs. Manage handle with cublasCreate(), cublasDestroy().
- Pass and return scalar values by reference to GPU memory.
- All functions return an error code.
- Rename cublasSetKernelStream() to cublasSetStream() for consistency with other CUDA libraries.

CUSPARSE

- New library for sparse basic linear algebra
- Conversion routines for dense, COO, CSR and CSC formats
- Optimized sparse matrix-vector multiplication
- Building block for sparse linear solvers

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \alpha \begin{bmatrix} 1.0 & & & \\ 2.0 & 3.0 & & \\ & & 4.0 & \\ 5.0 & & 6.0 & 7.0 \end{bmatrix} \begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{bmatrix} + \beta \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

CUDA Libm features

High performance and high accuracy implementation:

- C99 compatible math library, plus extras
- Basic ops: x+y, x*y, x/y, 1/x, sqrt(x), FMA (IEEE-754 accurate in single, double)
- Exponentials: exp, exp2, log, log2, log10, ...
- Trigonometry: sin, cos, tan, asin, acos, atan2, sinh, cosh, asinh, acosh, ...
- Special functions: lgamma, tgamma, erf, erfc
- Utility: fmod, remquo, modf, trunc, round, ceil, floor, fabs, ...
- Extras: rsqrt, rcbrt, exp10, sinpi, sincos, erfinv, erfcinv, ...

Improvements

- Continuous enhancements to performance and accuracy

CUDA 3.1 erfinvf (single precision)
accuracy
5.43 ulp → 2.69 ulp
performance
1.7x faster than CUDA 3.0

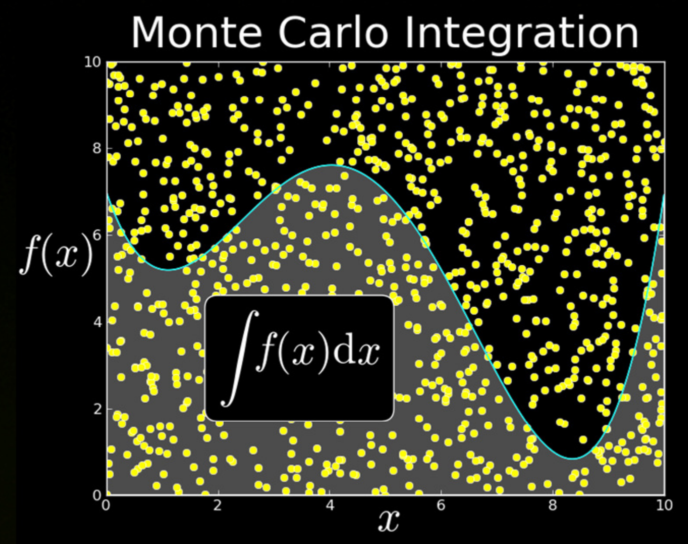
$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

CUDA 3.2 1/x (double precision)
performance
1.8x faster than CUDA 3.1

Double-precision division, rsqrt(), erfc(), & sinh() are all >~30% faster on Fermi

CURAND Library

- **Library for generating random numbers**



- **Features:**
  - **XORWOW pseudo-random generator**
  - **Sobol' quasi-random number generators**
  - **Host API for generating random numbers in bulk**
  - **Inline implementation allows use inside GPU functions/kernels**
  - **Single- and double-precision, uniform, normal and log-normal distributions**

CURAND Features

- Pseudo-random numbers
  George Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14), 2003. Available at http://www.jstatsoft.org/v08/i14/paper.
- Quasi-random 32-bit and 64-bit Sobol' sequences with up to 20,000 dimensions.
- Host API: call kernel from host, generate numbers on GPU, consume numbers on host or on GPU.
- GPU API: generate and consume numbers during kernel execution.

CURAND use

1. **Create a generator:**
   curandCreateGenerator()
2. **Set a seed:**
   curandSetPseudoRandomGeneratorSeed()
3. **Generate the data from a distribution:**
   curandGenerateUniform()/(curandGenerateUniformDouble(): Uniform
   curandGenerateNormal()/cuRandGenerateNormalDouble(): Gaussian
   curandGenerateLogNormal/curandGenerateLogNormalDouble(): Log-Normal
4. **Destroy the generator:**
   curandDestroyGenerator()

Example CURAND Program: Host API

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand.h>

main()
{
    int i, n = 100;
    curandGenerator_t gen;
    float *devData, *hostData;

    /* Allocate n floats on host */
    hostData = (float *)calloc(n, sizeof(float));

    /* Allocate n floats on device */
    cudaMalloc((void **)&devData, n * sizeof(float));

    /* Create pseudo-random number generator */
    curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
    /* Set seed */
    curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);
    /* Generate n floats on device */
    curandGenerateUniform(gen, devData, n);
    /* Copy device memory to host */
    cudaMemcpy(hostData, devData, n * sizeof(float),
                          cudaMemcpyDeviceToHost);

    /* Show result */
    for(i = 0; i < n; i++) {
        printf("%1.4f ", hostData[i]);
    }
    printf("\n");

    /* Cleanup */
    curandDestroyGenerator(gen);
    cudaFree(devData);
    free(hostData);

    return 0;
}
```

Example CURAND Program: Run on CPU

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand.h>

main()
{
    int i, n = 100;
    curandGenerator_t gen;
    float *hostData;

    /* Allocate n floats on host */
    hostData = (float *)calloc(n, sizeof(float));

    /* Create pseudo-random number generator */
    curandCreateGeneratorHost(&gen,
            CURAND_RNG_PSEUDO_DEFAULT);

    /* Set seed */
    curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);
    /* Generate n floats on host */
    curandGenerateUniform(gen, hostData, n);

    /* Show result */
    for(i = 0; i < n; i++) {
        printf("%1.4f ", hostData[i]);
    }
    printf("\n");

    /* Cleanup */
    curandDestroyGenerator(gen);
    free(hostData);

    return 0;
}
```

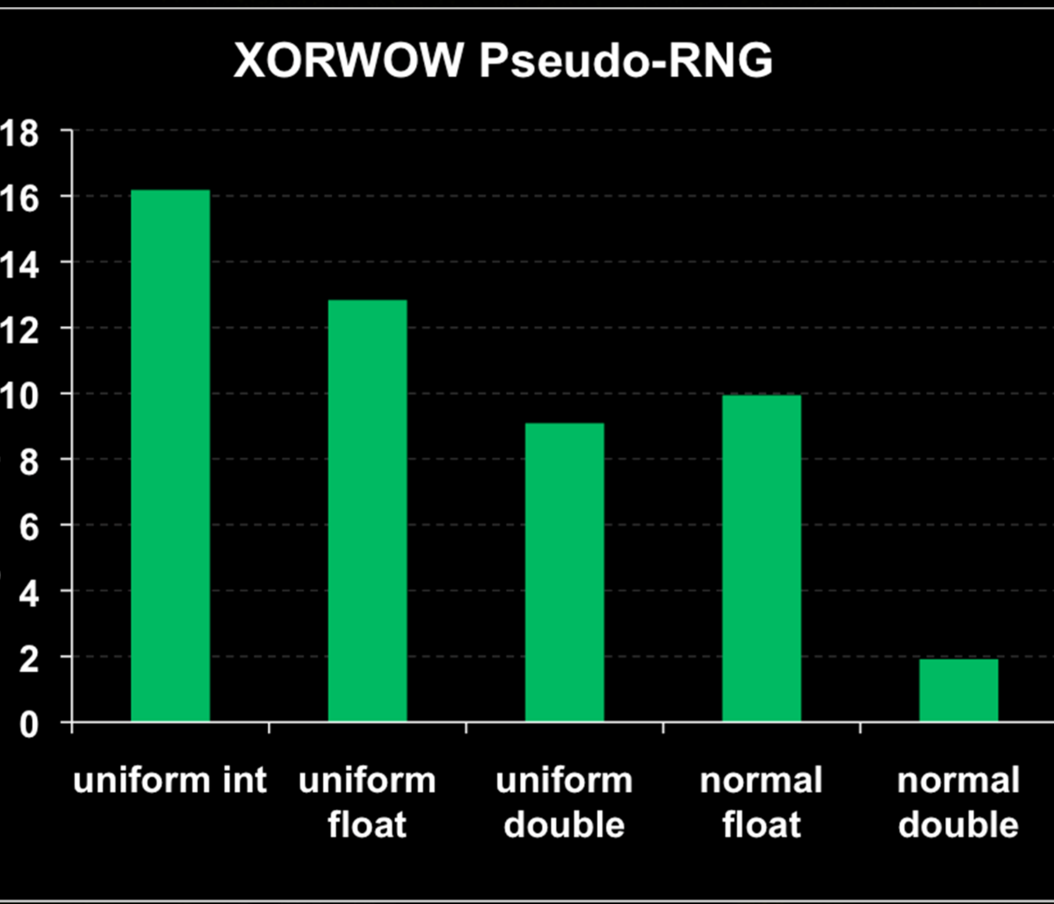
Example CURAND Program: Device API

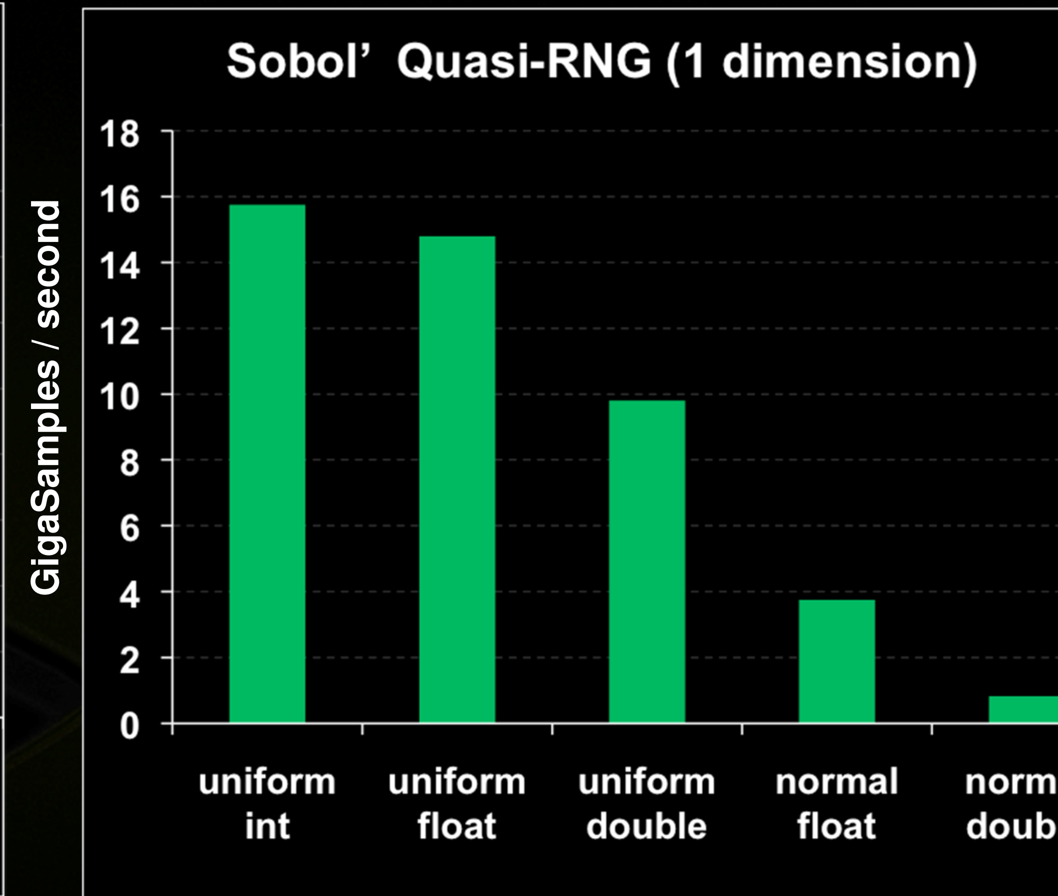
```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand_kernel.h>

__global__ void setup_kernel(curandState *state)
{
        int id = threadIdx.x + blockIdx.x * 64;
        /* Each thread gets same seed,
          a different sequence number, no offset */
        curand_init(1234, id, 0, &state[id]);
}

__global__ void generate_kernel(curandState *state, int *result)
{
        int id = threadIdx.x + blockIdx.x * 64;
        int count = 0;
        unsigned int x;
        /* Copy state to local memory for efficiency */
        curandState localState = state[id];

        /* Generate pseudo-random unsigned ints */
        for(int n = 0; n < 100000; n++) {
                x = curand(&localState);
                /* Check if low bit set */
                if(x & 1) count++;
        }

        /* Copy state back to global memory */
        state[id] = localState;

        /* Store results */
        result[id] += count;
}
```

Example CURAND Program: Device API

```
main()

    int i, total;
    curandState *devStates;
    int *devResults, *hostResults;

    /* Allocate space for results on host */
    hostResults = (int *)calloc(64 * 64, sizeof(int));

    /* Allocate space for results on device */
    cudaMalloc((void **)&devResults, 64 * 64 * sizeof(int));

    /* Set results to 0 */
    cudaMemset(devResults, 0, 64 * 64 * sizeof(int));

    /* Allocate space for prng states on device */
    cudaMalloc((void **)&devStates, 64 * 64 * sizeof(curandState));

    /* Setup prng states */
    setup_kernel<<<64, 64>>>(devStates);

    /* Generate and use pseudo-random */
    for(i = 0; i < 10; i++) {
        generate_kernel<<<64, 64>>>(devStates, devResults);
    }

    /* Copy device memory to host */
    cudaMemcpy(hostResults, devResults, 64 * 64 * sizeof(int),
                   cudaMemcpyDeviceToHost);

    /* Show result */
    total = 0;
    for(i = 0; i < 64 * 64; i++) {
        total += hostResults[i];
    }
    printf("Fraction with low bit set was %10.13f\n",
    (float)total / (64.0f * 64.0f * 100000.0f * 10.0f));

    /* Cleanup */
    cudaFree(devStates);
    cudaFree(devResults);
    free(hostResults);

    return 0;
}
```

CURAND Performance





Performance may vary based on OS version and motherboard configuration

CURAND 3.2, NVIDIA C2050 (Fermi), ECC on

NVIDIA Performance Primitives (NPP)

- C library of functions (primitives)
  - well optimized
  - low level API:
    - easy integration into existing code
    - algorithmic building blocks
  - actual operations execute on CUDA GPUs
- Approximately 350 image processing functions
- Approximately 100 signal processing functions

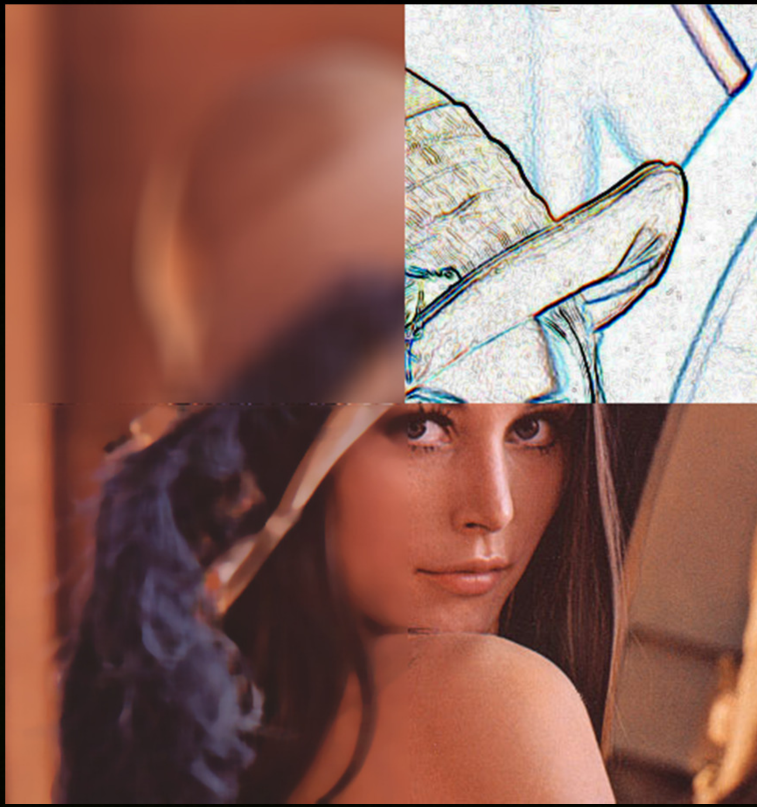


Image Processing Primitives

- Data exchange & initialization
  - Set, Convert, CopyConstBorder, Copy, Transpose, SwapChannels
- Arithmetic & Logical Ops
  - Add, Sub, Mul, Div, AbsDiff
- Threshold & Compare Ops
  - Threshold, Compare
- Color Conversion
  - RGB To YCbCr (& vice versa), ColorTwist, LUT_Linear
- Filter Functions
  - FilterBox, Row, Column, Max, Min, Dilat
    Erode, SumWindowColumn/Row
- Geometry Transforms
  - Resize , Mirror, WarpAffine/Back/Quad, WarpPerspective/Back/Quad
- Statistics
  - Mean, StdDev, NormDiff, MinMax, Histogram, SqrIntegral, RectStdDev
- Segmentation
  - Graph Cut

Thrust

- A template library for CUDA
  - Mimics the C++ STL
- Containers
  - Manage memory on host and device: thrust::host_vector<T>, thrust:device_vector<T>
  - Help avoid common errors
- Iterators
  - Know where data lives
  - Define ranges: d_vec.begin()
- Algorithms
  - Sorting, reduction, scan, etc: thrust::sort()
  - Algorithms act on ranges and support general types and operators

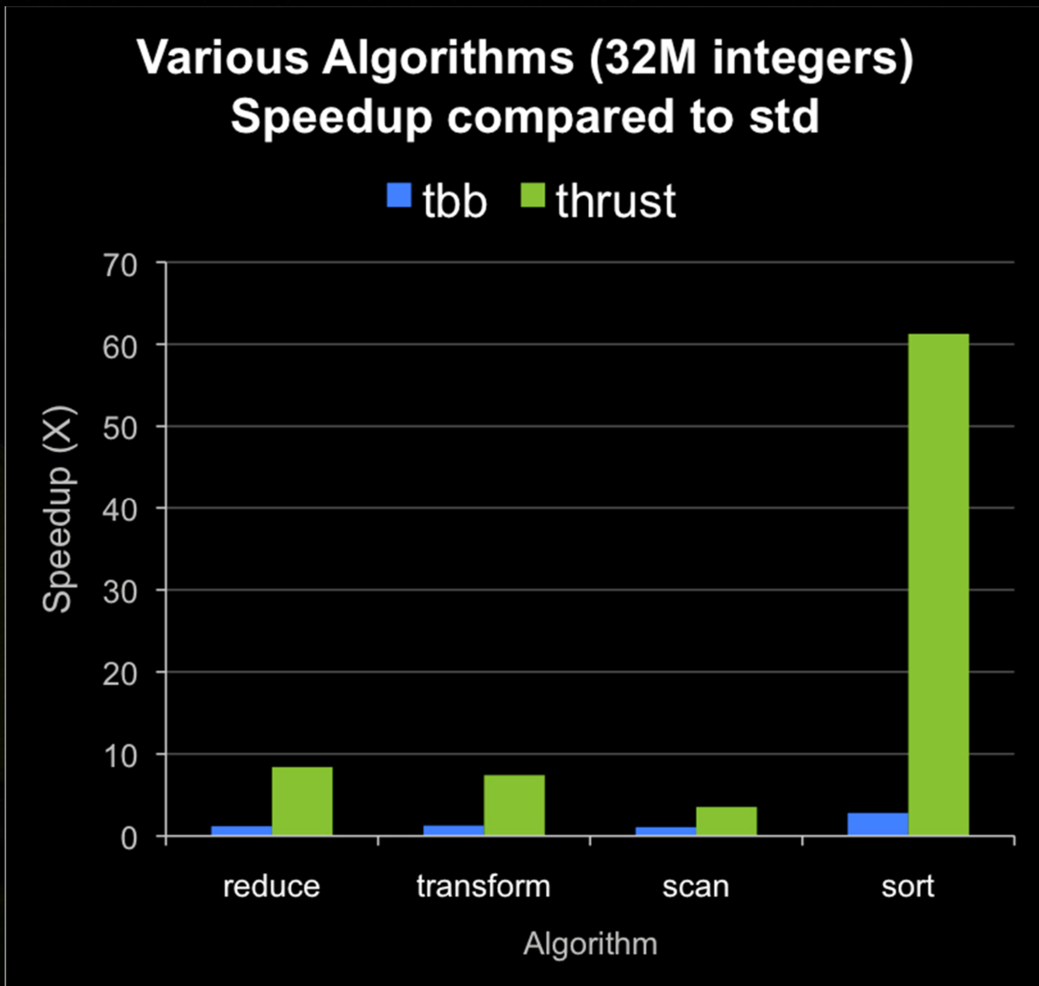
Thrust Example

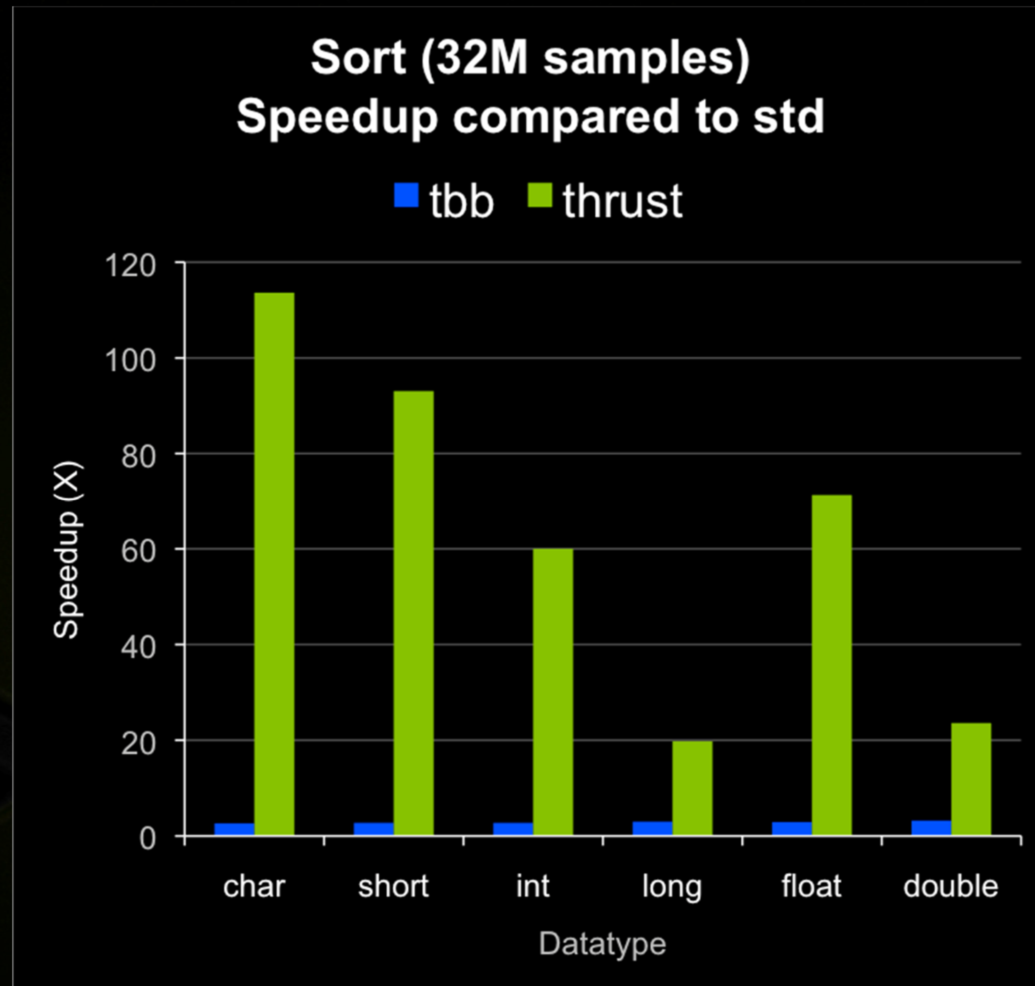
```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <cstdlib.h>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

Algorithms

- Elementwise operations
  - `for_each`, `transform`, `gather`, `scatter` …
- Reductions
  - `reduce`, `inner_product`, `reduce_by_key` …
- Prefix-Sums
  - `inclusive_scan`, `inclusive_scan_by_key` …
- Sorting
  - `sort`, `stable_sort`, `sort_by_key` …

Thrust Algorithm Performance



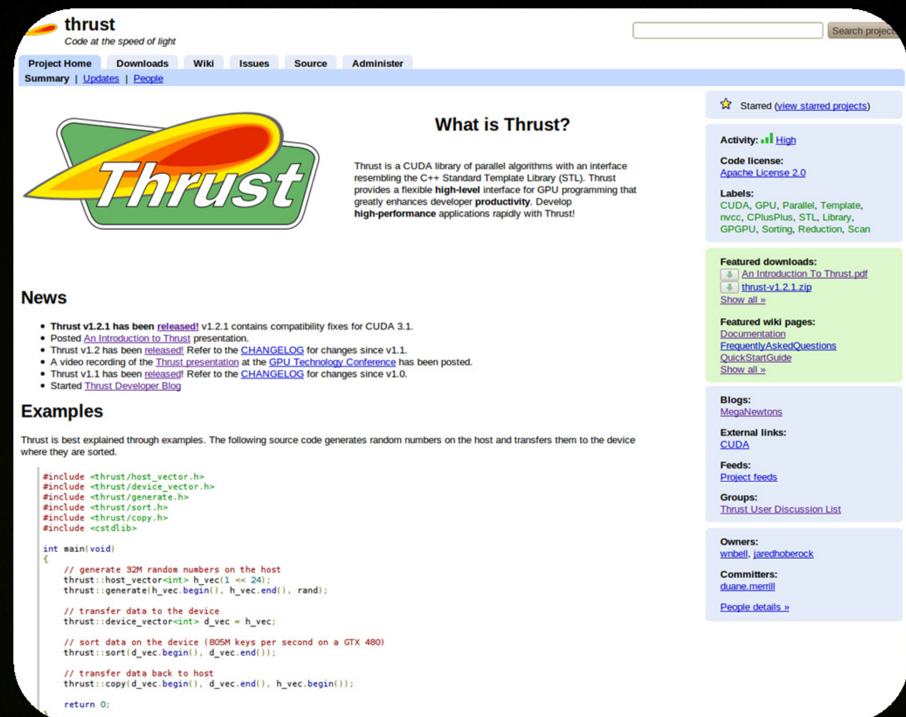


* Thrust 4.0, NVIDIA Tesla C2050 (Fermi)

* Core i7 950 @ 3.07GHz

Thrust on Google Code

- Quick Start Guide
- Examples
- Documentation
- Mailing List (thrust-users)
- http://code.google.com/p/thrust/



3rd Party Libraries

Library	Uses
MAGMA	Linear Algebra (Dense)
CULA Tools*	Linear Algebra (Dense)
CUSP	Linear Algebra (Sparse)
CUDPP	Parallel Primitives Library
OpenCurrent	CFD
NAG*	Computational Finance
And many more…	

*Commercial Libraries

Questions?