CUDA Libraries and CUDA Fortran

Massimiliano Fatica
NVIDIA Corporation

NVIDIA CUDA Libraries

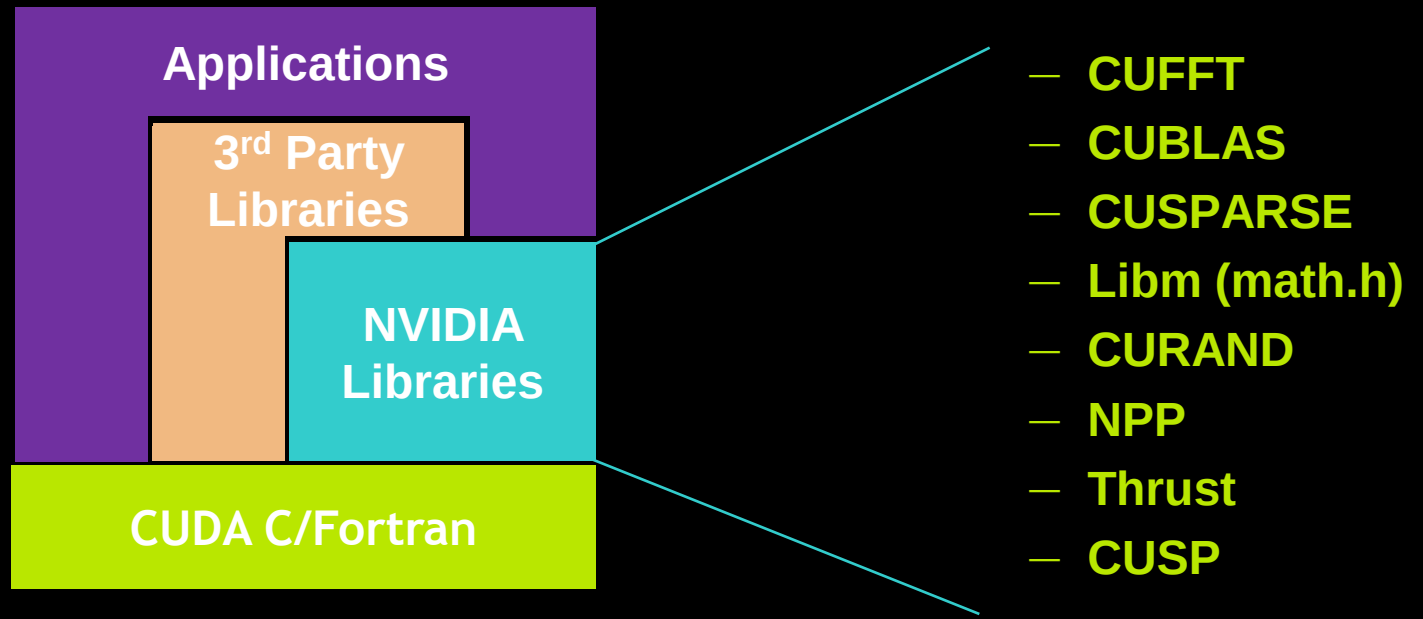
CUDA Toolkit includes several libraries:

- **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 : Template Library**

Several open source and commercial* libraries:

- **MAGMA: Linear Algebra**
- **CULA Tools*: Linear Algebra**
- **CUSP: Sparse Linear Solvers**
- **NAG*: Computational Finance**
- **OpenVidia: Computer Vision**
- **OpenCurrent: CFD**
- **……..**

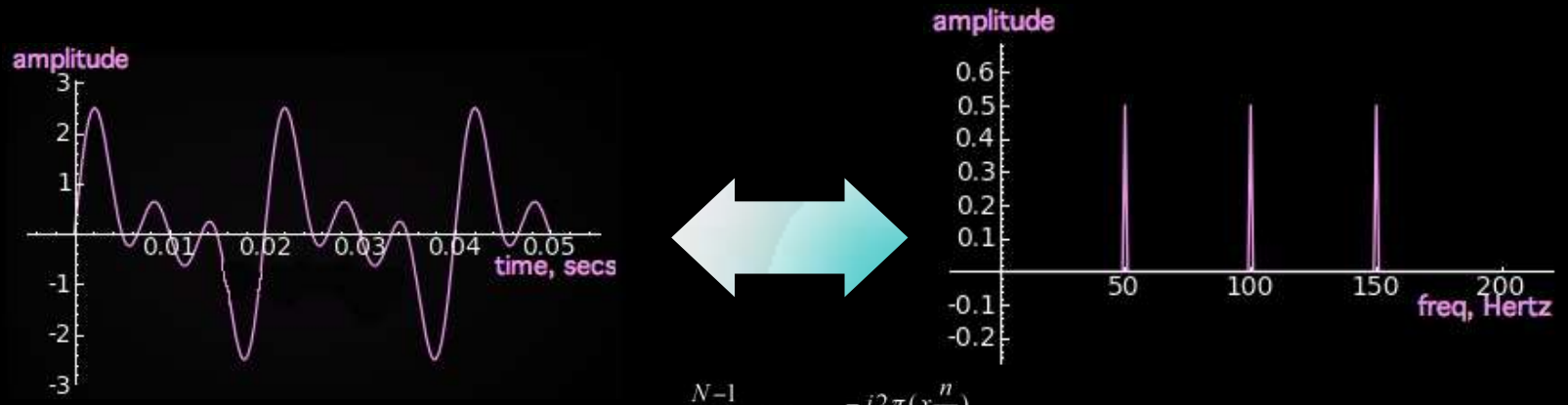
NVIDIA CUDA Libraries



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

CUFFT Library

CUFFT is a GPU based Fast Fourier Transform library

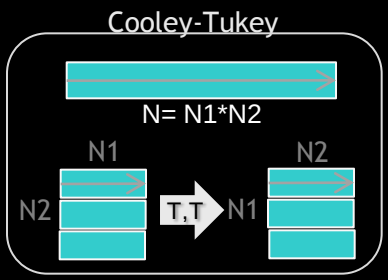


$$F(x) = \sum_{n=0}^{N-1} f(n) e^{-j2\pi(x\frac{n}{N})}$$

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

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

Step 1 – Allocate space on GPU memory

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

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

Step 4 – Destroy plan, free GPU memory

Code example:

```
#define NX 256
#define NY 128

cufftHandle plan;
cufftComplex *idata, *odata;
cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*NY);
…
/* Create a 2D FFT plan. */
 cufftPlan2d(&plan, NX,NY, CUFFT_C2C);

 /* Use the CUFFT plan to transform the signal out of place. */
 cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);

/* Inverse transform the signal in place.
 Different pointers to input and output arrays implies out of place transformation  */
 cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);
….
 /* Destroy the CUFFT plan. */
   cufftDestroy(plan);

   cudaFree(idata), cudaFree(odata);
```

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

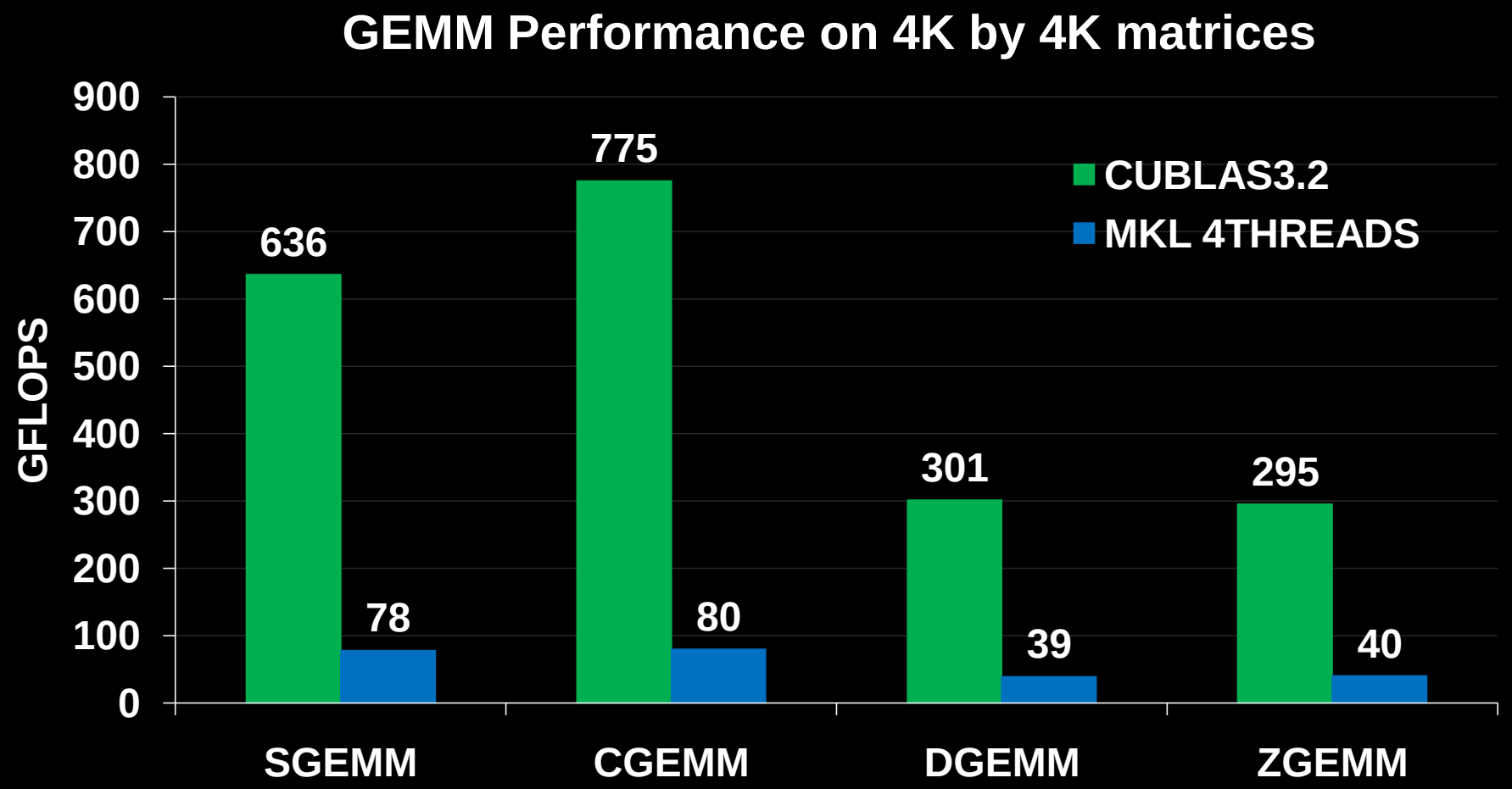
SGEMM example (NON-THUNKING)

```
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
```

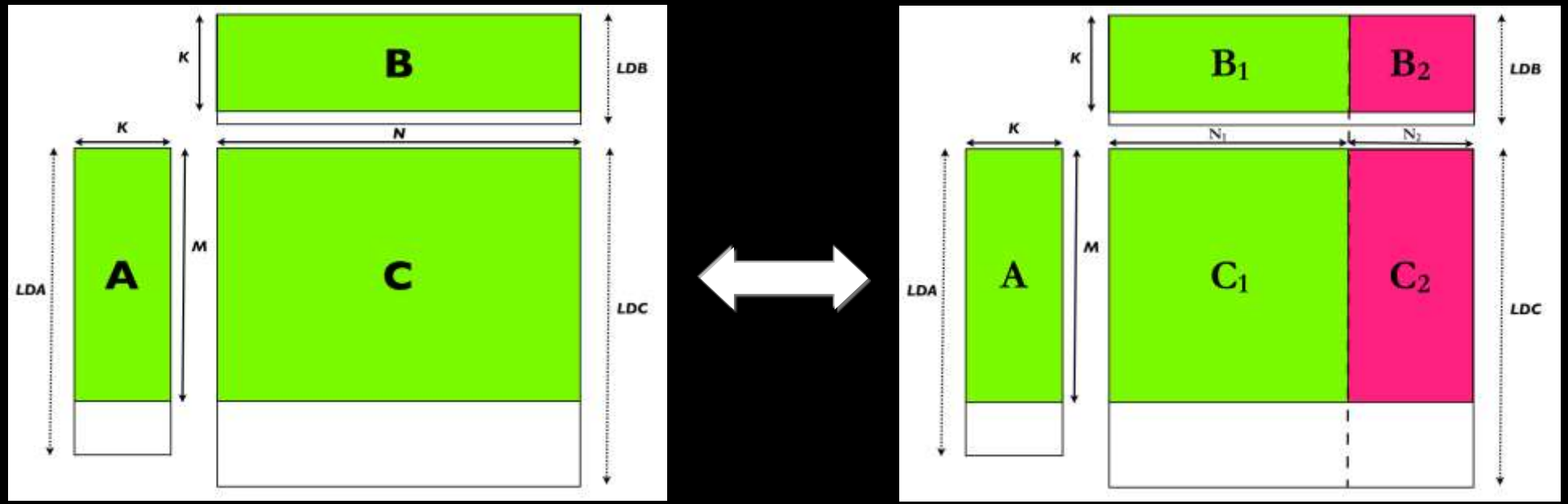
GEMM Performance



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

Using CPU and GPU concurrently



DGEMM(A,B,C) = DGEMM(A,B1,C1) U DGEMM(A,B2,C2)

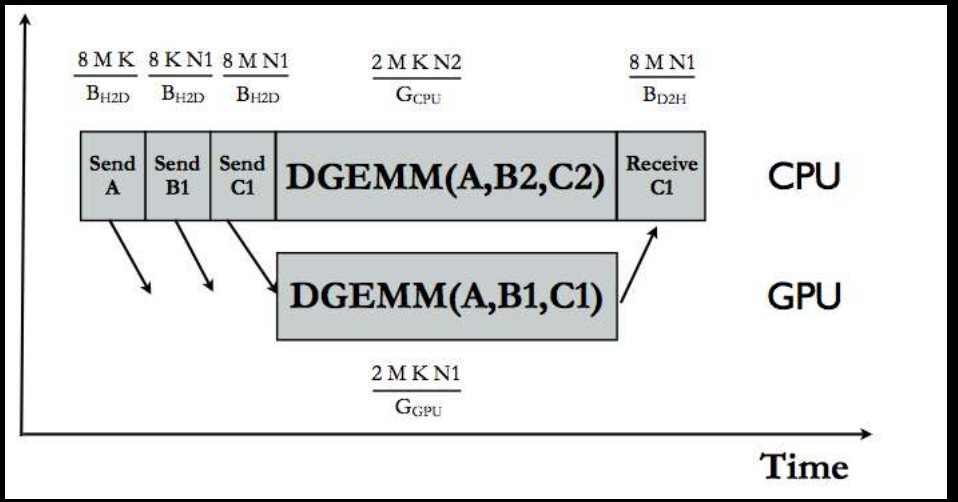
(GPU) (CPU)

The idea can be extended to multi-GPU configuration and to handle huge matrices

Find the optimal split, knowing the relative performances of the GPU and CPU cores on DGEMM

Overlap DGEMM on CPU and GPU

```
// Copy A from CPU memory to GPU memory devA
   status = cublasSetMatrix (m, k    , sizeof(A[0]), A, lda, devA, m_gpu);
// Copy B1  from CPU memory to GPU memory devB
   status = cublasSetMatrix (k ,n_gpu, sizeof(B[0]), B, ldb, devB, k_gpu);
// Copy C1  from CPU memory to GPU memory devC
   status = cublasSetMatrix (m, n_gpu, sizeof(C[0]), C, ldc, devC, m_gpu);

// Perform  DGEMM(devA,devB,devC) on GPU
// Control immediately return to CPU
   cublasDgemm('n', 'n', m, n_gpu, k, alpha, devA, m,devB, k, beta, devC, m);

// Perform  DGEMM(A,B2,C2) on CPU
   dgemm('n','n',m,n_cpu,k, alpha, A, lda,B+ldb*n_gpu, ldb, beta,C+ldc*n_gpu, ldc);

// Copy devC  from GPU memory to CPU memory C1
   status = cublasGetMatrix (m, n, sizeof(C[0]), devC, m, C, *ldc);
```



Using CUBLAS, it is very easy to express the workflow in the diagram

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

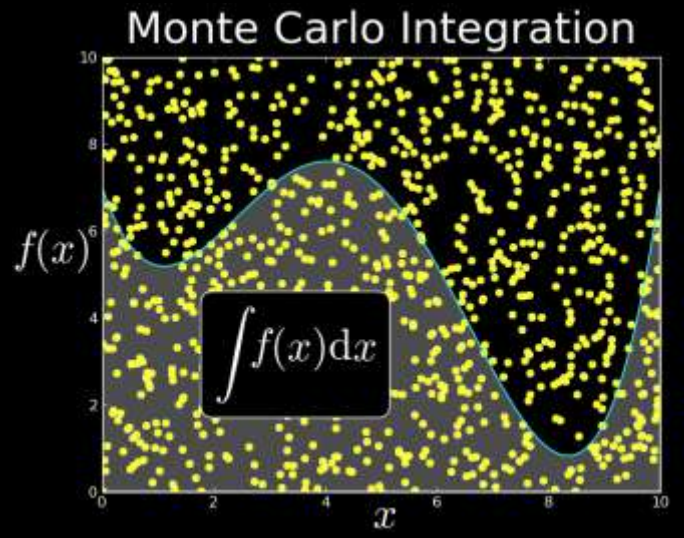
$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

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;
}
```

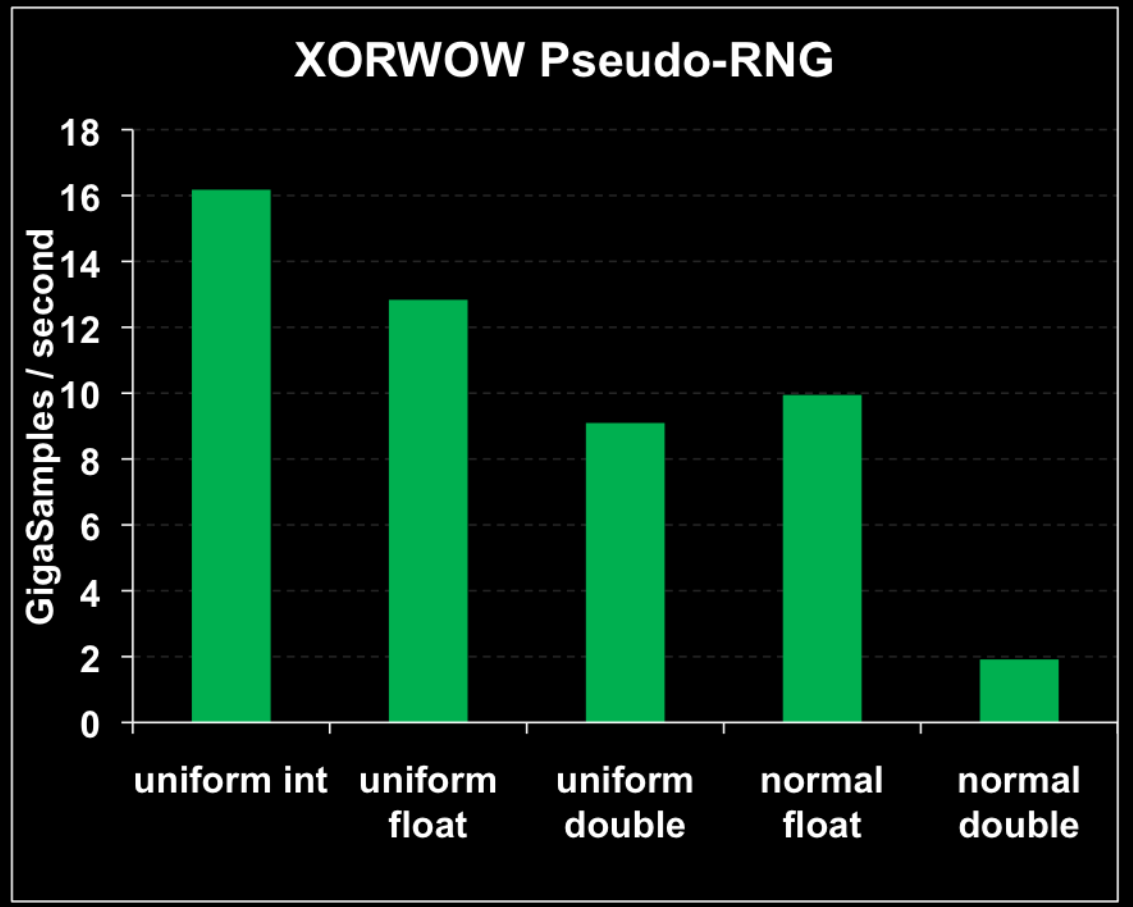
Example CURAND Program: Device API

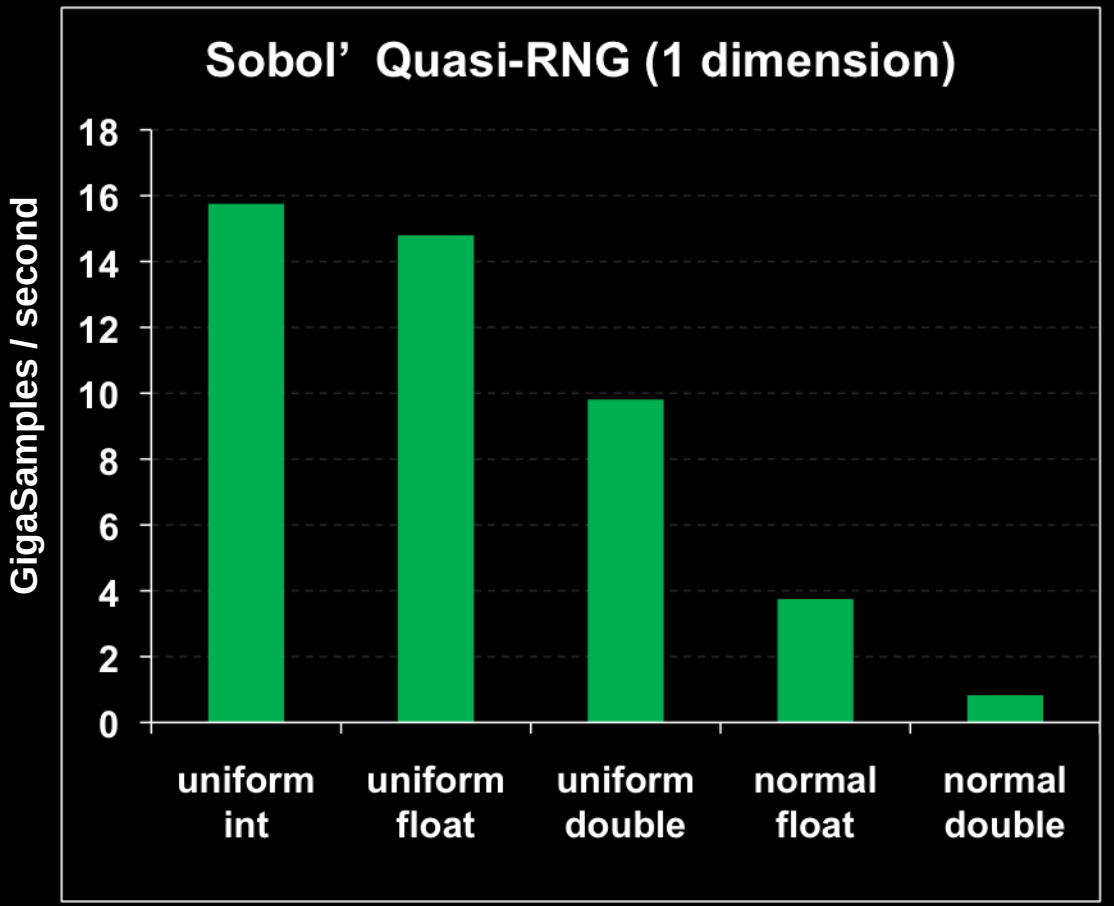
```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand_kernel.h>

__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

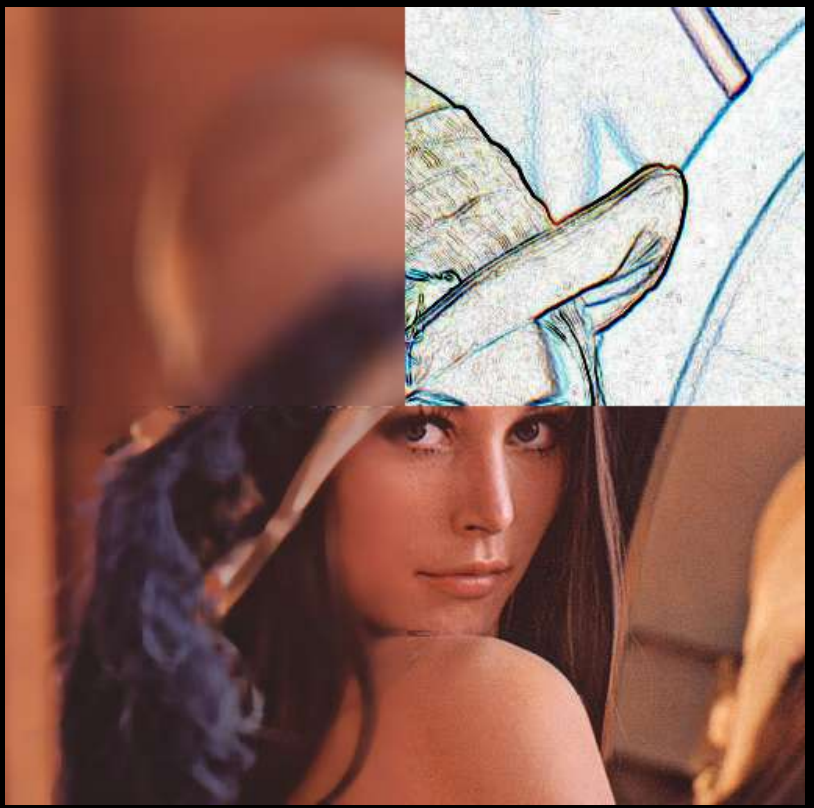


Image Processing Primitives

- 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, Dilate, 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` …

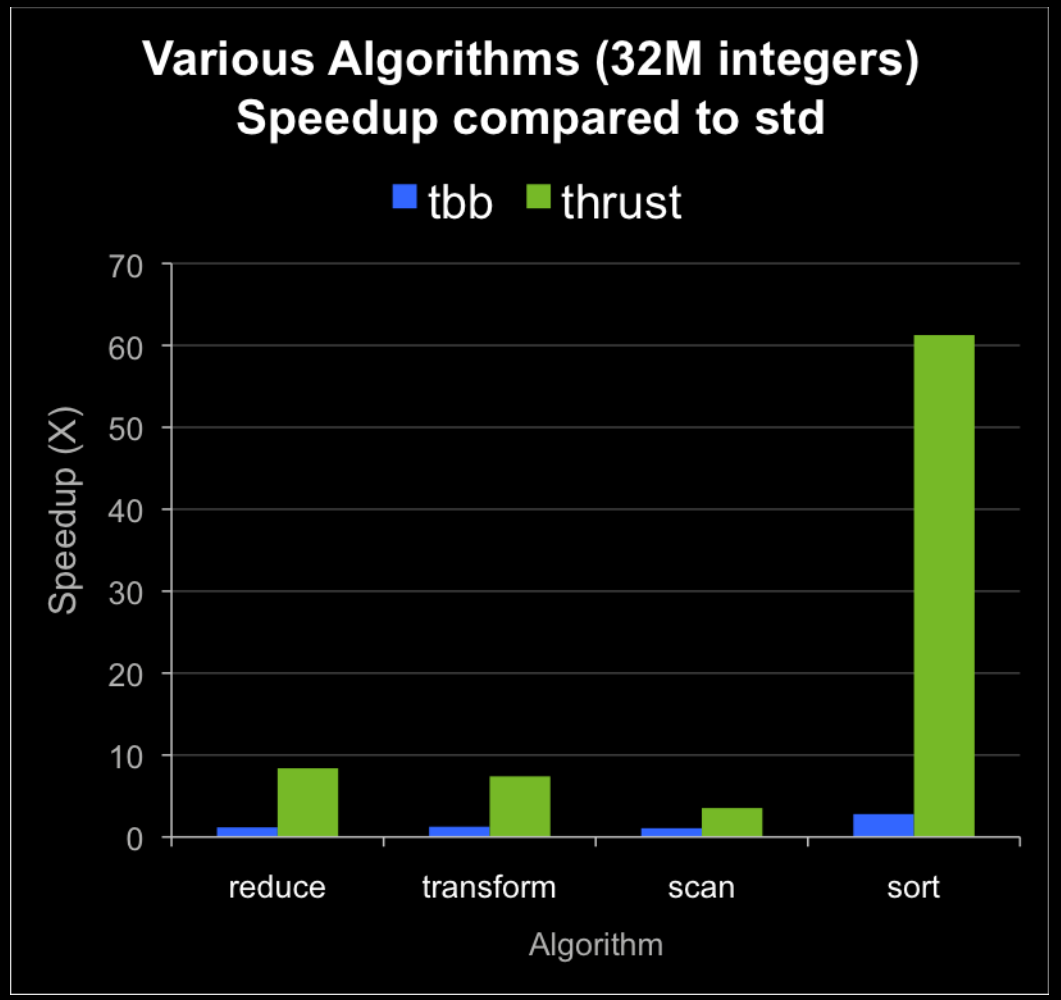
- **Prefix-Sums**

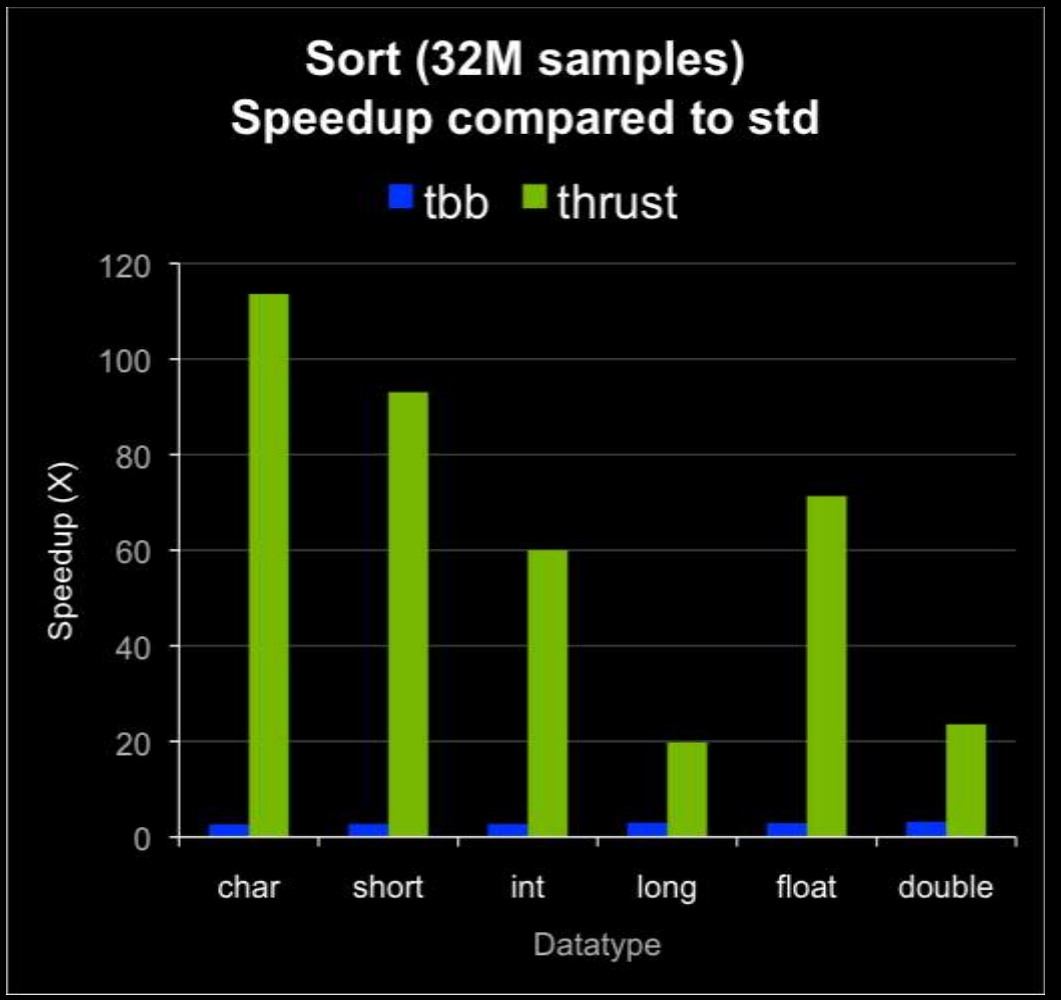
  `inclusive_scan`, `inclusive_scan_by_key` …

- **Sorting**

  `sort`, `stable_sort`, `sort_by_key` …

Thrust Algorithm Performance





* Thrust 4.0, NVIDIA Tesla C2050 (Fermi)

* Core i7 950 @ 3.07GHz

Interoperability (from Thrust to C/CUDA)

- Convert iterators to raw pointers

```
// allocate device vector
thrust::device_vector<int> d_vec(4);

// obtain raw pointer to device vector's memory
int * ptr = thrust::raw_pointer_cast(&d_vec[0]);

// use ptr in a CUDA C kernel
my_kernel<<< N / 256, 256 >>>(N, ptr);

// Note: ptr cannot be dereferenced on the host!
```

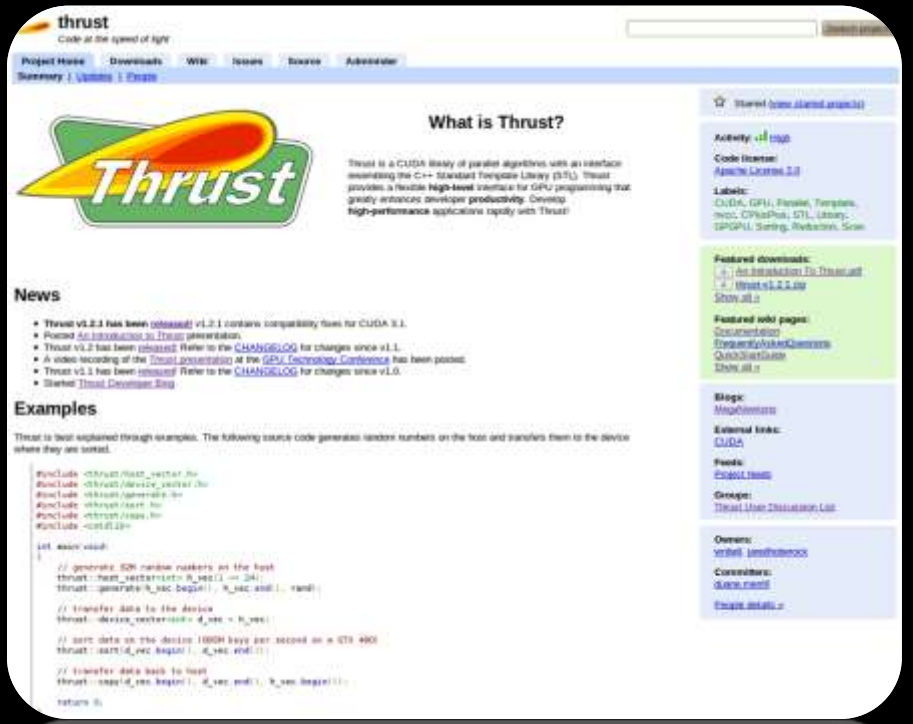
Interoperability (from C/CUDA to Thrust)

- Wrap raw pointers with `device_ptr`

```
// raw pointer to device memory
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof int));

// wrap raw pointer with a device_ptr
device_ptr<int> dev_ptr(raw_ptr);

// use device_ptr in thrust algorithms
fill(dev_ptr, dev_ptr + N, (int) 0);

// access device memory through device_ptr
dev_ptr[0] = 1;

// free memory
cudaFree(raw_ptr);
```

Thrust on Google Code

- Quick Start Guide
- Examples
- Documentation
- Mailing List (thrust-users)



CUDA Fortran

- PGI / NVIDIA collaboration
- Same CUDA programming model as CUDA-C with Fortran syntax
- Strongly typed – variables with device-type reside in GPU memory
- Use standard allocate, deallocate
- Copy between CPU and GPU with assignment statements:

```
GPU_array = CPU_array
```

- Kernel loop directives (CUF Kernels) to parallelize loops with device data

CUDA Fortran example

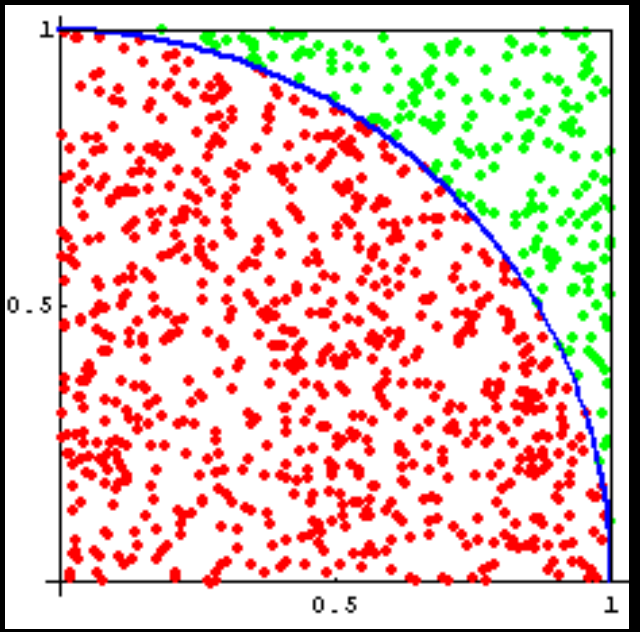
```
program gpu_example
 use cudafor
 real, allocatable :: cpu_array(:,:)
 real, allocatable, device :: gpu_array(:,:)
 type(dim3):: grid_size, block_size
! Use standard allocate for CPU and GPU arrays
 allocate(cpu_array(n,m), gpu_array(n,m))
 call initialize(cpu_array)
! Move data with simple assignment
 gpu_array = cpu_array
! Call CUDA kernel
  …
  block_size=dim3(16,16,1)
  call gpu_func<<<grid_size, block_size>>> ( gpu_array, n,m )
  cpu_array = gpu_array
  deallocate(cpu_array, gpu_array)
end program
```

CUDA Fortran example

```
attributes(global) subroutine gpu_func(gpu_array, n, m)
real:: gpu_array(:,:)
integer, value:: n, m
integer:: i,j
real::  p_ref=1.0

! Indices start from 1
i = threadIdx%x+ (blockIdx%x -1) * blockDim%x
j = threadIdx%y+ (blockIdx%y -1) * blockDim%y
If ( i <= n .and. j<=m)
            gpu_array(i,j) = SQRT( 2)*( gpu_array(i,j) + p_ref )
end if

end subroutine gpu_func
```

Computing π with CUDA Fortran

$$\pi = 4 * (\Sigma \text{ red points})/(\Sigma \text{ points})$$



Simple example:

- **Generate random numbers (CURAND)**
- **Compute sum using of kernel loop directive**
- **Compute sum using two stages reduction with Cuda Fortran kernels**
- **Compute sum using single stage reduction with Cuda Fortran kernel**
- **Accuracy**

CUDA Libraries from CUDA Fortran

- CUBLAS, CUFFT and CURAND have C interfaces
- Use F90 interfaces and ISO C Binding to call them directly from CUDA Fortran

```
interface curandGenerateUniform

!curandGenerateUniform(curandGenerator_t generator, float *outputPtr, size_t num);
subroutine curandGenerateUniform(generator, odata, numele) bind(C,name='curandGenerateUniform')
 use iso_c_binding
 integer(c_size_t),value:: generator
!pgi$ ignore_tr odata
 real(c_float), device:: odata(*)
 integer(c_size_t),value:: numele
end subroutine curandGenerateUniform

!curandGenerateUniformDouble(curandGenerator_t generator, double *outputPtr, size_t num);
subroutine curandGenerateUniformDouble(generator, odata, numele) bind(C,name='curandGenerateUniformDouble')
 use iso_c_binding
 integer(c_size_t),value:: generator
!pgi$ ignore_tr odata
 real(c_double), device:: odata(*)
 integer(c_size_t),value:: numele
end subroutine curandGenerateUniformDouble

end interface curandGenerateUniform
```

Computing π with CUF kernel

```
! Compute pi using a Monte Carlo method
program compute_pi
use precision
use cudafor   ! CUDA Fortran runtime
use curand    ! CURAND interface
implicit none
real(fp_kind), allocatable, pinned:: hostData(:)
real(fp_kind), allocatable, device:: deviceData(:)
real(fp_kind):: pival
integer :: inside_cpu,inside, i, iter, Nhalf
integer(kind=8) :: gen, N, seed=1234

N=2000 ! Define how many numbers we want to generate
Nhalf=N/2
! Allocate arrays on CPU and GPU
 allocate(hostData(N), deviceData(N))
! Create pseudonumber generator
 call curandCreateGenerator(gen, CURAND_RNG_PSEUDO_DEFAULT)
! Set seed
 call curandSetPseudoRandomGeneratorSeed( gen, seed)

! Generate N floats or double on device
call curandGenerateUniform(gen, deviceData, N)

! Copy the data back to CPU to check result later
 hostData=deviceData
! Perform the test on GPU using CUF kernel
 inside=0
!$cuf kernel do <<<*,*>>>
 do i=1,Nhalf
  if( (deviceData(i)**2+deviceData(i+Nhalf)**2) <= 1._fp_kind ) inside=inside+1
 end do
! Perform the test on CPU
 inside_cpu=0
 do i=1,Nhalf
  if( (hostData(i)**2+hostData(i+Nhalf)**2) <= 1._fp_kind) inside_cpu=inside_cpu+1
 end do
! Check the results
 if (inside_cpu .ne. inside) print *,"Mismatch between CPU/GPU", inside_cpu,inside
! Print the value of pi and the error
 pival= 4._fp_kind*real(inside,fp_kind)/real(Nhalf,fp_kind)
 print"(t3,a,i10,a,f10.8,a,e11.4)","Samples=",Nhalf,"  Pi=", pival,"  Error=", &
          abs(pival-2.0_fp_kind*asin(1.0_fp_kind))
! Deallocate data on CPU and GPU
 deallocate(hostData,deviceData)
! Destroy the generator
 call curandDestroyGenerator(gen)
end program compute_pi
```

Computing π

pgf90 -Mcuda=3.2 -O3 -Mpreprocess -o pi_gpu precision_module.cuf curand_module.cuf pi.cuf -lcurand

```
Compute pi  in single precision (seed 1234)
 Samples=        10000  Pi=3.11120009  Error= 0.3039E-01
 Samples=       100000  Pi=3.13632011  Error= 0.5273E-02
 Samples=      1000000  Pi=3.14056396  Error= 0.1029E-02
 Samples=     10000000  Pi=3.14092445  Error= 0.6683E-03
 Samples=    100000000  Pi=3.14158082  Error= 0.1192E-04
```

```
Compute pi  in single precision (seed 1234567)
 Samples=        10000  Pi=3.16720009  Error= 0.2561E-01
 Samples=       100000  Pi=3.13919997  Error= 0.2393E-02
 Samples=      1000000  Pi=3.14109206  Error= 0.5007E-03
 Samples=     10000000  Pi=3.14106607  Error= 0.5267E-03
  Mismatch between CPU/GPU     78534862     78534859
 Samples=    100000000  Pi=3.14139414  Error= 0.1986E-03
```

Where is the error coming from?

```
if( (    hostData(i)**2+   hostData(i+Nhalf)**2) <= 1._fp_kind) inside_cpu=inside_cpu+1  (CPU)
if( (deviceData(i)**2+deviceData(i+Nhalf)**2) <= 1._fp_kind ) inside=inside+1              (GPU)
```

–Sum of the point inside the circle is done with integers (no issues due to floating point arithmetic)

–Computation of the distance from the origin (x*x+y*y), no special functions just + and *

GPU accuracy

- FERMI GPUs are IEEE-754 compliant, both for single and double precision
- Support for Fused Multiply-Add instruction (IEEE 754-2008)
- Results with FMA could be different* from results without FMA
- In CUDA Fortran is possible to toggle FMA on/off with a compiler switch:
  -Mcuda=nofma
- Extremely useful to compare results to “golden” CPU output
- FMA is being supported in future CPUs

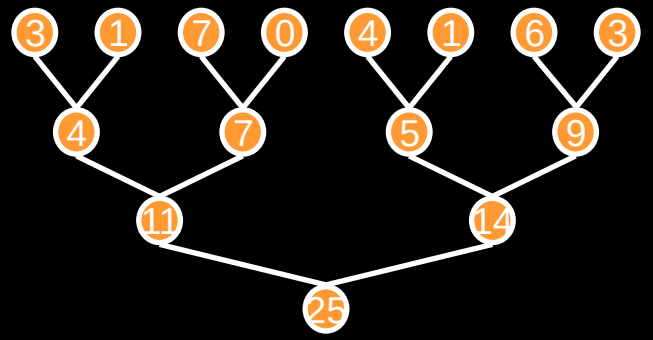
```
Compute pi  in single precision (seed=1234567 FMA disabled)
  Samples=        10000  Pi=3.16720009  Error= 0.2561E-01
  Samples=       100000  Pi=3.13919997  Error= 0.2393E-02
  Samples=      1000000  Pi=3.14109206  Error= 0.5007E-03
  Samples=     10000000  Pi=3.14106607  Error= 0.5267E-03
  Samples=    100000000  Pi=3.14139462  Error= 0.1981E-03
```

*GPU results with FMA are identical to CPU if operations are done in double precision

Reductions on GPU

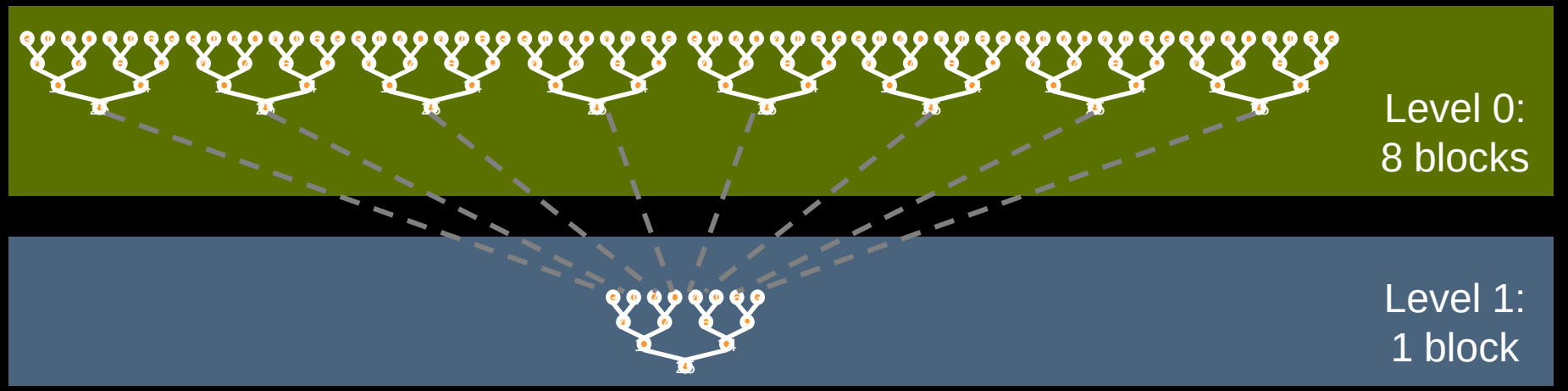
- Need to use multiple blocks

- Need to use multiple threads in a block:

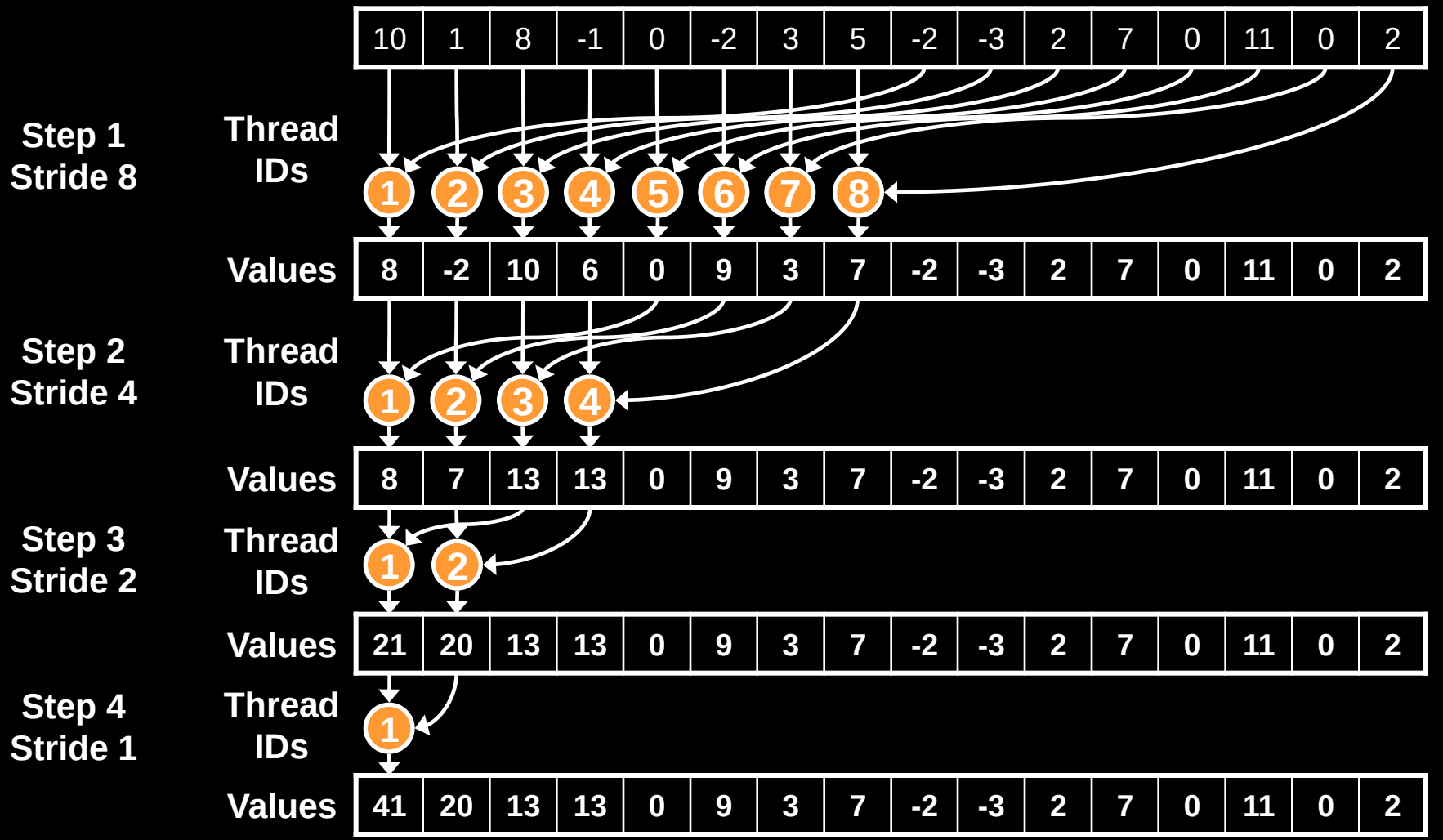


-No global synchronization:

two stages approach, same code for both stages



Parallel Reduction: Sequential Addressing



Computing π with CUDA Fortran kernels (1/2)

```
attributes(global) subroutine partial_sum(input,partial,N)
 real(fp_kind) :: input(N)
 integer :: partial(256)
 integer, shared, dimension(256) :: psum
 integer(kind=8),value :: N
 integer :: i,index, inext,interior

 index=threadIdx%x+(BlockIdx%x-1)*BlockDim%x
! Check if the point is inside the circle and increment local counter
 interior=0
 do i=index,N/2,BlockDim%x*GridDim%x
  if( (input(i)**2+input(i+N/2)**2) <= 1._fp_kind) interior=interior+1
 end do

 ! Local reduction per block
 index=threadIdx%x
 psum(index)=interior
 call syncthreads()

 inext=blockDim%x/2
 do while ( inext >=1 )
  if (index <=inext) psum(index)=psum(index)+psum(index+inext)
  inext = inext /2
  call syncthreads()
 end do

! Each block writes back its partial sum
 if (index == 1) partial(BlockIdx%x)=psum(1)
end subroutine
```

```
! Compute the partial sums with 256 blocks of 512 threads
  call partial_sum<<<256,512,512*4>>>(deviceData,partial,N)
! Compute the final sum with 1 block of 256 threads
  call final_sum<<<1,256,256*4>>>(partial,inside_gpu)
```

Computing π with CUDA Fortran kernels (2/2)

```
attributes(global) subroutine final_sum(partial,nthreads,total)
 integer, intent(in) :: partial(nthreads)
 integer, intent(out) :: total
 integer, shared :: psum(*)
 integer :: index, inext

 index=threadIdx%x

! load partial sums in shared memory
 psum(index)=partial(index)
 call syncthreads()

 inext=blockDim%x/2
 do while ( inext >=1 )
  if (index <=inext) psum(index)=psum(index)+psum(index+inext)
  inext = inext /2
  call syncthreads()
 end do

! First thread has the total sum, writes it back to global memory
 if (index == 1) total=psum(1)
end subroutine
```

Computing π with atomic lock

Instead of storing back the partial sum:

```
! Each block writes back its partial sum
if (index == 1) partial(BlockIdx%x)=psum(1)
```

use atomic lock to ensure that one block at the time updates the final sum:

```
if (index == 1) then
  do while ( atomiccas(lock,0,1) == 1) !set lock
  end do
  partial(1)=partial(1)+psum(1) ! atomic update of partial(1)
  call threadfence()            ! Wait for memory transaction to be  visible to all the other threads
  lock =0                       ! release lock
 end if
```

```
partial(1)=0
call sum<<<64,256,256*4>>>(deviceData,partial,N)
inside=partial(1)
```

Calling Thrust from CUDA Fortran

C wrapper for Thrust: csort.cu

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

extern "C" {
//Sort for integer arrays
void sort_int_wrapper( int *data, int N)
{
  // Wrap raw pointer with a device_ptr
    thrust::device_ptr<int> dev_ptr(data);
 // Use device_ptr in Thrust sort algorithm
    thrust::sort(dev_ptr, dev_ptr+N);
}

//Sort for single precision arrays
void sort_float_wrapper( float *data, int N)
{
   thrust::device_ptr<float> dev_ptr(data);
   thrust::sort(dev_ptr, dev_ptr+N);
}

//Sort for double precision arrays
void sort_double_wrapper( double *data, int N)
{
   thrust::device_ptr<double> dev_ptr(data);
   thrust::sort(dev_ptr, dev_ptr+N);
}
}
```

Calling Thrust from CUDA Fortran

Fortran interface to C wrapper using ISO C Bindings

```
module thrust

 interface thrustsort

 subroutine sort_int( input,N) bind(C,name="sort_int_wrapper")
  use iso_c_binding
  integer(c_int),device:: input(*)
  integer(c_int),value:: N
 end subroutine

 subroutine sort_float( input,N) bind(C,name="sort_float_wrapper")
  use iso_c_binding
  real(c_float),device:: input(*)
  integer(c_int),value:: N
 end subroutine

 subroutine sort_double( input,N) bind(C,name="sort_double_wrapper")
  use iso_c_binding
  real(c_double),device:: input(*)
  integer(c_int),value:: N
 end subroutine

 end interface
end module
```

CUDA Fortran sorting with Thrust

```
program testsort
use thrust
real, allocatable :: cpuData(:)
real, allocatable, device :: gpuData(:)
integer:: N=10
!Allocate CPU and GPU arrays
  allocate(cpuData(N),gpuData(N))
!Fill the host array with random data
 do i=1,N
   cpuData(i)=random(i)
 end do
! Print unsorted data
  print *, cpuData
! Send data to GPU
 gpuData = cpuData
!Sort the data
 call thrustsort(gpuData,N)
!Copy the result back
 cpuData = gpuData
! Print sorted data
  print *, cpuData
!Deallocate arrays
 deallocate(cpuData,gpuData)
end program testsort
```

```
nvcc -c -arch sm_20 csort.cu
pgf90 -rc=rc4.0 -Mcuda=cc20 -O3 -o testsort thrust_module.cuf testsort.cuf csort.o
```

```
 $ ./testsort
Before sorting 4.1630346E-02 0.9124327 0.7832350 0.6540373 100.0000 0.3956419 0.2664442 0.13724658.0488138E-03 0.8788511
After sorting 8.0488138E-03 4.1630346E-02 0.1372465 0.26644420.3956419 0.6540373 0.7832350 0.87885110.9124327 100.0000
```

CUDA Fortran sorting with Thrust

```
program timesort
use cudafor
use thrust
real, allocatable :: cpuData(:)
real, allocatable, device :: gpuData(:)
integer:: N=100000000,istat
! cuda events for elapsing
type ( cudaEvent ) :: startEvent , stopEvent
real :: time, random
!Allocate CPU and GPU arrays
  allocate(cpuData(N),gpuData(N))
!Fill the host array with random data
 do i=1,N
   cpuData(i)=random(i)
 end do
```

```
! Create events
istat = cudaEventCreate ( startEvent )
istat = cudaEventCreate ( stopEvent )
! Send data to GPU
gpuData = cpuData
!Sort the data
istat = cudaEventRecord ( startEvent , 0)
 call thrustsort(gpuData,N)
istat = cudaEventRecord ( stopEvent , 0)
istat = cudaEventSynchronize ( stopEvent )
istat = cudaEventElapsedTime ( time , startEvent , stopEvent )
!Copy the result back
cpuData = gpuData
print *," Sorted array in:",time," (ms)"
print *,"After sorting", cpuData(1:5),cpuData(N-4:N)
!Deallocate arrays
 deallocate(cpuData,gpuData)
end program timesort
```

$ ./timesort

```
Sorting array of     100000000  single precision
Sorted array in:    194.6642      (ms)
After sorting   7.0585919E-09   1.0318221E-08   1.9398616E-08   3.1738640E-08  4.4078664E-08   0.9999999       0.9999999       1.000000     1.000000       1.000000
```