CUDA Libraries and CUDA Fortran

Massimiliano Fatica
NVIDIA Corporation
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 - OpenVidia: Computer Vision
- CULA Tools*: Linear Algebra - OpenCurrent: CFD
- CUSP: Sparse Linear Solvers .......
- NAG*: Computational Finance
NVIDIA CUDA Libraries

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

Applications

3rd Party

NVIDIA Libraries

CUDA C/Fortran
CUFFT Library

CUFFT is a GPU based Fast Fourier Transform library
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: \(\text{IFFT}(\text{FFT}(A)) = \text{len}(A) \cdot A\)
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:

```c
#define NX 256
#define NY 128

cuftHandle plan;
cuftComplex *idata, *odata;
cudaMalloc((void**)&idata, sizeof(cuftComplex)*NX*NY);
cudaMalloc((void**)&odata, sizeof(cuftComplex)*NX*NY);
...

/* Create a 2D FFT plan. */
cuftPlan2d(&plan, NX, NY, CUFFT_C2C);

/* Use the CUFFT plan to transform the signal out of place. */
cuftExecC2C(plan, idata, odata, CUFFT_FORWARD);

/* Inverse transform the signal in place.
Different pointers to input and output arrays implies out of place transformation */
cuftExecC2C(plan, odata, odata, CUFFT_INVERSE);

....

/* Destroy the CUFFT plan. */
cuftDestroy(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
  - **Level 1 (vector,vector):** $O(N)$
    - AXPY: $y = \alpha x + y$
    - DOT: $\text{dot} = x \cdot y$
  - **Level 2 (matrix,vector):** $O(N^2)$
    - Vector multiplication by a General Matrix: GEMV
    - Triangular solver: TRSV
  - **Level 3 (matrix,matrix):** $O(N^3)$
    - General Matrix Multiplication: GEMM
    - Triangular Solver: TRSM
- Following BLAS convention, CUBLAS uses column-major storage
- Function naming convention: cublas + BLAS name
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
#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();
print(" 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*lda] = (float)(rand()/(float)RAND_MAX;
            b[i*ldb] = (float)(rand()/(float)RAND_MAX;
            c[i*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, 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 );
    
td1 = t1.tv_sec - t4.tv_sec + 1.0e-6 *( t4.tv_usec - t1.tv_usec);
    td2 = 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(:,,:),allocated:: 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/lib64 -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(1:,1:), allocatable :: A(1:,1:), B(1:,1:), C(1:,1:)
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
Using CPU and GPU concurrently

\[ \text{DGEMM}(A, B, C) = \text{DGEMM}(A, B_1, C_1) \cup \text{DGEMM}(A, B_2, C_2) \]

\begin{align*}
\text{(GPU)} & \quad \text{(CPU)} \\
\text{The idea can be extended to multi-GPU configuration and to handle huge matrices} \\
\text{Find the optimal split, knowing the relative performances of the GPU and CPU cores on DGEMM}
\end{align*}
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.
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, ...
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 use

1. Create a generator:
   \texttt{curandCreateGenerator()}

2. Set a seed:
   \texttt{curandSetPseudoRandomGeneratorSeed()}

3. Generate the data from a distribution:
   \texttt{curandGenerateUniform()/(curandGenerateUniformDouble()): Uniform}
   \texttt{curandGenerateNormal()/cuRandGenerateNormalDouble(): Gaussian}
   \texttt{curandGenerateLogNormal/cuRandGenerateLogNormalDouble(): Log-Normal}

4. Destroy the generator:
   \texttt{curandDestroyGenerator()}
Example CURAND Program: Host API

```c
#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("%.4f ", hostData[i]);
    }
    printf("\n");

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

    return 0;
}
```
Example CURAND Program: Run on CPU

```c
#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 *)malloc(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;
```
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` ...
Interoperability (from Thrust to C/CUDA)

- Convert iterators to raw pointers

```cpp
// 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!
```
Wrap raw pointers with `device_ptr`

```c
// 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);
```
Introduction to CUDA Fortran
Introduction

- CUDA is a scalable model for parallel computing

- CUDA Fortran is the Fortran analog to CUDA C
  - Program has host and device code similar to CUDA C
  - Host code is based on the runtime API
  - Fortran language extensions to simplify data management

- Co-defined by NVIDIA and PGI, implemented in the PGI Fortran compiler
CUDA Programming

• Heterogeneous programming model
  - CPU and GPU are separate devices with separate memory spaces
  - Host code runs on the CPU
    • Handles data management for both host and device
    • Launches kernels which are subroutines executed on the GPU
  - Device code runs on the GPU
    • Executed by many GPU threads in parallel
  - Allows for incremental development
Heterogeneous Programming

- Host = CPU and its memory
- Device = GPU and its memory

- Typical code progression
  - Allocate memory on host and device
  - Transfer data from host to device
  - Execute kernel (device computation)
  - Transfer result from device to host
  - Deallocate memory
Data Transfers

program copyData
    use cudafor
    implicit none
    integer, parameter :: n = 256
    real :: a(n), b(n)
    real, device :: a_d(n), b_d(n)

    a = 1.0
    a_d = a
    b_d = a_d
    b = b_d

    if (all(a == b)) &
        write(*,*) 'Test Passed'
end program copyData
program copyData
  use cudafor
  implicit none
  integer, parameter :: n = 256
  real :: a(n), b(n)
  real, device :: a_d(n), b_d(n)

  a = 1.0
  a_d = a
  b_d = a_d
  b = b_d

  if (all(a == b)) &
      write(*,*) 'Test Passed'
end program copyData
Data Transfers

```fortran
program copyData
  use cudafor
  implicit none
  integer, parameter :: n = 256
  real :: a(n), b(n)
  real, device :: a_d(n), b_d(n)

  a = 1.0
  a_d = a
  b_d = a_d
  b = b_d

  if (all(a == b)) &
    write(*,*) 'Test Passed'
end program copyData
```
Data Transfers

program copyData
    use cudafor
    implicit none
    integer, parameter :: n = 256
    real :: a(n), b(n)
    real, device :: a_d(n), b_d(n)

    a = 1.0
    a_d = a
    b_d = a_d
    b = b_d

    if (all(a == b)) &
        write(*,*) 'Test Passed'
end program copyData
Data Transfers

program copyData
  use cudafor
  implicit none
  integer, parameter :: n = 256
  real :: a(n), b(n)
  real, device :: a_d(n), b_d(n)
  
a = 1.0
  a_d = a
  b_d = a_d
  b = b_d
  
  if (all(a == b)) &
    write(*,*) 'Test Passed'
end program copyData
module simpleOps_m
contains
    subroutine inc(a, b)
imPLICIT none
    integer :: a(:)
    integer :: b
    integer :: i, n

    n = size(a)
do i = 1, n
      a(i) = a(i)+b
  enddo

end subroutine inc
end module simpleOps_m

program incTest
use simpleOps_m
implicit none
integer :: b, n = 256
integer, allocatable:: a(:)
allocate (a(n))
a = 1 ! array assignment
b = 3
call inc(a, b)
if (all(a == 4)) &
  write(*,*) 'Test Passed'
deallocate(a)
end program incTest
program incTest

use simpleOps_m
implicit none
integer :: b, n = 256
integer, allocatable:: a(:)

allocate (a(n))
a = 1 ! array assignment
b = 3

call inc(a, b)

if (all(a == 4)) &
   write(*,*) 'Test Passed'
deallocate(a)

end program incTest

CUDA Fortran

program incTest

use cudafor
use simpleOps_m
implicit none
integer :: b, n = 256
integer, allocatable :: a(:)
integer, allocatable, device :: a_d(:)

allocate (a(n), a_d(n))
a = 1
b = 3

a_d = a
call inc<<<1,n>>>(a_d, b)
a = a_d

if (all(a == 4)) &
   write(*,*) 'Test Passed'
deallocate(a, a_d)

end program incTest
### CUDA Fortran - Device Code

<table>
<thead>
<tr>
<th>F90</th>
<th>CUDA Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>module simpleOps_m</td>
<td>module simpleOps_m</td>
</tr>
<tr>
<td>contains</td>
<td>contains</td>
</tr>
<tr>
<td>subroutine inc(a, b)</td>
<td>subroutine inc(a, b)</td>
</tr>
<tr>
<td>implicit none</td>
<td>implicit none</td>
</tr>
<tr>
<td>integer :: a(:)</td>
<td>integer :: a(:)</td>
</tr>
<tr>
<td>integer :: b</td>
<td>integer, value :: b</td>
</tr>
<tr>
<td>integer :: i, n</td>
<td>integer :: i</td>
</tr>
<tr>
<td>n = size(a)</td>
<td>i = threadIdx%x</td>
</tr>
<tr>
<td>do i = 1, n</td>
<td>a(i) = a(i)+b</td>
</tr>
<tr>
<td>a(i) = a(i)+b</td>
<td></td>
</tr>
<tr>
<td>enddo</td>
<td></td>
</tr>
<tr>
<td>end subroutine inc</td>
<td>end subroutine inc</td>
</tr>
<tr>
<td>end module simpleOps_m</td>
<td>end module simpleOps_m</td>
</tr>
</tbody>
</table>
Extending to Larger Arrays

- Previous example works with small arrays

  call inc<<<1,n>>>(a_d,b)

  - Limit of $n=1024$ (Fermi) or $n=512$ (pre-Fermi)

- For larger arrays, change the first execution parameter ($<<<1,n>>>$)
Execution Model

**Software**

- Thread
- Thread Block

**Hardware**

- Thread Processor
- Multiprocessor

**Threads**

- Threads are executed by thread processors
- Thread blocks are executed on multiprocessors
- Thread blocks do not migrate
- Several concurrent thread blocks can reside on a multiprocessor

**Grid**

- A kernel is launched on a device as a grid of thread blocks
Execution Configuration

• Execution configuration specified in host code

\[
\text{call inc} \langle\langle \text{blocksPerGrid}, \text{threadsPerBlock} \rangle\rangle (a_d, b)
\]

• Previous example used a single thread block

\[
\text{call inc} \langle\langle 1, n \rangle\rangle (a_d, b)
\]

• Multiple thread blocks

\[
tPB = 256 \\
\text{call inc} \langle\langle \text{ceiling(real(n)/tPB)}, tPB \rangle\rangle (a_d, b)
\]
Mapping Arrays to Thread Blocks

- call inc<<<3,4>>>(a_d, b)

\[
\text{blockDim}_x = 4 \\
\text{blockIdx}_x \\
\text{threadIdx}_x
\]

\[
(b\text{ blockIdx}_x - 1) \times \text{blockDim}_x + \text{threadIdx}_x
\]

1 2 3 4 5 6 7 8 9 10 11 12
program incTest
  use cudafor
  use simpleOps_m
  implicit none
  integer, parameter :: n = 1024*1024
  integer, parameter :: tPB = 256
  integer :: a(n), b
  integer, device :: a_d(n)

  a = 1
  b = 3

  a_d = a
  call inc<<<ceiling(real(n)/tPB),tPB>>>(a_d, b)
  a = a_d

  if (all(a == 4)) then
    write(*,*) 'Success'
  endif
end program incTest
Built-in Variables for Device Code

• Predefined variables in device subroutines
  - Grid and block dimensions - \texttt{gridDim}, \texttt{blockDim}
  - Block and thread indices - \texttt{blockIdx}, \texttt{threadIdx}
  - Of type \texttt{dim3}

  \begin{verbatim}
  type (dim3)
    integer (kind=4) :: x, y, z
  end type
  \end{verbatim}

  - \texttt{blockIdx} and \texttt{threadIdx} fields have unit offset

  \begin{verbatim}
  1 <= blockIdx%x <= gridDim%x
  \end{verbatim}
module simpleOps_m
contains
  attributes(global) subroutine inc(a, b)
  implicit none
  integer :: a(:)
  integer, value :: b
  integer :: i, n
  i = (blockIdx%x-1)*blockDim%x + threadIdx%x
  n = size(a)
  if (i <= n) a(i) = a(i)+ b
end subroutine inc
end module simpleOps_m
Multidimensional Example - Host

Execution Configuration

call inc<<<blocksPerGrid, threadsPerBlock>>>(a_d,b)

- Grid dimensions in blocks (\texttt{blocksPerGrid}) and block dimensions in threads (\texttt{threadsPerBlock}) can be either \texttt{integer} or \texttt{dim3}

type (dim3)
\begin{verbatim}
  integer (kind=4) :: x, y, z
end type
\end{verbatim}
program incTest
    use cudafor
    use simpleOps_m
    implicit none
    integer, parameter :: nx=1024, ny=512
    real :: a(nx,ny), b
    real, device :: a_d(nx,ny)
    type(dim3) :: grid, tBlock

    a = 1; b = 3

    tBlock = dim3(32,8,1)
    grid = dim3(ceiling(real(nx)/tBlock%x), ceiling(real(ny)/tBlock%y), 1)
    a_d = a
    call inc<<<grid,tBlock>>>(a_d, b)
    a = a_d

    write(*,*) 'Max error: ', maxval(abs(a-4))
end program incTest
module simpleOps_m
contains
  subroutine inc(a, b)
    implicit none
    real :: a(:,:)
    real, value :: b
    integer :: i, j

    i = (blockIdx%x-1)*blockDim%x + threadIdx%x
    j = (blockIdx%y-1)*blockDim%y + threadIdx%y

    if (i<=size(a,1) .and. j<=size(a,2)) &
      a(i,j) = a(i,j) + b
  end subroutine inc
end module simpleOps_m
Kernel Loop Directives (CUF Kernels)

• Automatic kernel generation and invocation of host code region (arrays in loops must reside on GPU)

```fortran
program incTest
  use cudafor
  implicit none
  integer, parameter :: n = 256
  integer :: a(n), b
  integer, device :: a_d(n)

  a = 1; b = 3; a_d = a

  !$cuf kernel <<<*,*>>>>
  do i = 1, n
    a_d(i) = a_d(i)+b
  enddo

  a = a_d
  if (all(a == 4)) write(*,*) 'Test Passed'
end program incTest
```
Kernel Loop Directives (CUF Kernels)

• Multidimensional arrays

    !$cuf kernel do(2) <<< *,* >>>
    do j = 1, ny
        do i = 1, nx
            a_d(i,j) = b_d(i,j) + c_d(i,j)
        enddo
    enddo

• Can specify parts of execution parameter

    !$cuf kernel do(2) <<<(*,*),(32,4)>>>

Reduction using CUF Kernels

• Compiler recognizes use of scalar reduction and generates one result

\[ rsum = 0.0 \]

\[ !$cuf \text{ kernel do } <<<*,*>>> \]
\[ \text{do } i = 1, nx \]
\[ \quad rsum = rsum + a_d(i) \]
\[ \text{enddo} \]
## Compute Capabilities

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Tesla</th>
<th>Fermi</th>
<th>Kepler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute capabilities</td>
<td>1.0</td>
<td>1.3</td>
<td>2.0</td>
</tr>
<tr>
<td>Double precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3D grids</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max # threads per block</td>
<td>512</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>Shared memory per MP</td>
<td>16Kb</td>
<td>48Kb (16/48, 48/16)</td>
<td>48Kb (32/32)</td>
</tr>
</tbody>
</table>

- All these values are returned by `cudaGetDeviceProperties`
- Target GPU can be specified with `-Mcuda=ccx.x`
Compilation

- **pgfortran** - PGI’s Fortran compiler
  - All source code with `.cuf` or `.CUF` is compiled as CUDA Fortran enabled automatically
  - Flag to target architecture (eg. `-Mcuda=cc20`)
  - `-Mcuda=emu` specifies emulation mode
  - Flag to target CUDA Toolkit version (eg. `-Mcuda=cuda4.0`)
  - `-Mcuda=fastmath` enables faster intrinsics (`__sinf()`)  
  - `-Mcuda=nofma` turns off fused multiply-add
  - `-Mcuda=maxregcount:<n>` limits register usage per thread
  - `-Mcuda=ptxinfo` prints memory usage per kernel

- **pgfortran** `-Mcuda` `-help` for a full list
Host-Device Transfers

• Host-device bandwidth is much lower than bandwidth within device
  - 8 GB/s peak (PCIe x16 Gen 2) vs. 177 GB/s peak (Tesla M2090)

• Minimize number of transfers
  - Intermediate data can be allocated, used, and deallocated without copying to host memory
  - Sometimes better to do low parallelism operations on the GPU if it avoids transfers to and from host
Page-Locked Data Transfers

**Pageable Data Transfer**
- **Device**
  - DRAM
- **Host**
  - Pageable memory
  - Pinned buffer

**Page-locked Data Transfer**
- **Device**
  - DRAM
- **Host**
  - Pinned memory
Page-Locked Data Transfers

• Page-locked or pinned host memory by declaration
  - Designated by \texttt{pinned} variable attribute
  - Must be \texttt{allocatable}

\begin{verbatim}
real, device :: a_d(N)
real, pinned, allocatable :: a(:)
allocate(a(N), STAT=istat, PINNED=pinnedFlag)
...
a_d = a
\end{verbatim}

• Tesla M2050/Nehalem
  - Pageable: \(\sim 3.5 \text{ GB/s}\)
  - Pinned: \(\sim 6 \text{ GB/s}\)
Overlapping Transfers and Computation

- Kernel launches are asynchronous, normal memory copies are blocking
  
  ```
  a_d = a  ! blocks on host until transfer completes
  call inc<<<g,b>>>(a_d, b)  ! Control returns immediately to CPU
  a = a_d  ! starts only after kernel completes
  ```

- Asynchronous and Stream APIs allow overlap of transfers with computation

- A stream is a sequence of operations that execute in order on the GPU
  - Operations in different (non-default) streams can be interleaved
  - Stream ID used as arguments to async transfers and kernel launches
Asynchronous Data Transfers

- Asynchronous host-device transfers return control immediately to CPU
  - `cudaMemcpyAsync(dst, src, nElements, stream)`
  - Requires pinned host memory

- Overlapping data transfer with CPU computation
  - default stream = 0

```c
istat = cudaMemcpyAsync(a_d, a_h, N, 0)
call kernel<<<grid, block>>>(a_d)
call cpuFunction(b)
```
Overlapping Transfers and Kernels

- Requires:
  - Pinned host memory
  - Kernel and transfer to use different non-zero streams

```fortran
integer (kind=cuda_stream_kind) :: stream1, stream2
...
istat = cudaStreamCreate(stream1)
istat = cudaStreamCreate(stream2)
istat = cudaMemcpyAsync(a_d, a_h, N, stream1)
call kernel<<<grid, block, 0, stream2>>>(b_d)
```

overlapped
GPU/CPU Synchronization

- **cudaDeviceSynchronize()**
  - Blocks until all previously issued operations on the GPU complete

- **cudaStreamSynchronize(stream)**
  - Blocks until all previously issued operations to `stream` complete

- **cudaStreamQuery(stream)**
  - Indicated whether `stream` is idle
  - Does not block CPU code
GPU/CPU Synchronization

- Stream-based using CUDA events
  - Events can be inserted into streams
  
  ```
  type (cudaEvent) :: event
  ...
  istat = cudaEventRecord(event, stream1)
  ```

  - Event is recorded when the GPU reaches it in the stream
    - Recorded = assigned a time stamp
    - Useful for timing code

- `cudaEventSynchronize(event)`
  - Blocks CPU until event is recorded
Shared Memory

- On-chip
- All threads in a block have access to same shared memory
- Used to reduce multiple loads of device data
- Used to accommodate coalescing
Matrix Transpose

attributes(global) subroutine transposeNaive(odata, idata)
  real, intent(out) :: odata(ny,nx)
  real, intent(in) :: idata(nx,ny)
  integer :: x, y

  x = (blockIdx%x-1) * blockDim%x + threadIdx%x
  y = (blockIdx%y-1) * blockDim%y + threadIdx%y
  odata(y,x) = idata(x,y)
end subroutine transposeNaive
Matrix Transpose - Shared Memory

```fortran
attributes(global) subroutine transposeCoalesced(odata, idata)
  real, intent(out) :: odata(ny,nx)
  real, intent(in) :: idata(nx,ny)
  real, shared :: tile(TILE_DIM, TILE_DIM)
  integer :: x, y

  x = (blockIdx%x-1)*blockDim%x + threadIdx%x
  y = (blockIdx%y-1)*blockDim%y + threadIdx%y
  tile(threadIdx%x, threadIdx%y) = idata(x,y)

  call syncthreads()

  x = (blockIdx%y-1)*blockDim%y + threadIdx%y
  y = (blockIdx%x-1)*blockDim%x + threadIdx%x
  odata(x,y) = tile(threadIdx%y, threadIdx%x)
end subroutine transposeCoalesced
```
Calling CUBLAS from CUDA Fortran

- Module which defines interfaces to CUBLAS from CUDA Fortran
  - use cublas

- Interfaces in three forms
  - Overloaded BLAS interfaces that take device array arguments
    - call saxpy(n, a_d, x_d, incx, y_d, incy)
  - Legacy CUBLAS interfaces
    - call cublasSaxpy(n, a_d, x_d, incx, y_d, incy)
  - Multi-GPU version (CUDA 4.0) that utilizes a handle h
    - istat = cublasSaxpy_v2(h, n, a_d, x_d, incx, y_d, incy)

- Mixing the three forms is allowed
Calling CUBLAS from CUDA Fortran

program cublasTest
  use cublas
  implicit none

  real, allocatable :: a(:,:),b(:,:),c(:,:)
  real, device, allocatable :: a_d(:,:),b_d(:,:),c_d(:,:)
  integer :: k=4, m=4, n=4
  real :: alpha=1.0, beta=2.0, maxError

  allocate(a(m,k), b(k,n), c(m,n), a_d(m,k), b_d(k,n), c_d(m,n))
  a = 1; a_d = a
  b = 2; b_d = b
  c = 3; c_d = c

  call cublasSgemm('N','N',m,n,k,alpha,a_d,m,b_d,k,beta,c_d,m) ! or sgemm(..)
  c=c_d
  write(*,*) 'Maximum error: ', maxval(abs(c-14.0))
  deallocate (a,b,c,a_d,b_d,c_d)

end program cublasTest
Calling Thrust from CUDA Fortran

C wrapper for Thrust: csort.cu

```c
#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);
    }
}
```
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
  end interface

  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

  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
end module
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
# 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)

$ ./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
```

Convolution Example

Perform convolution of a stack of 2D arrays in frequency space

1. Send arrays to GPU memory
2. Transform arrays to frequency domain using 2D FFT from CUFFT
3. Perform point-wise complex multiplication and scaling \( (FFT(IFFT(A))=\text{len}(A)\times A) \)
4. Transform result back to physical domain
5. Send result back to CPU memory
Naive approach

Smarter approach

Optimal approach

NOT TO SCALE
program driver
use cudafor
use cufft
implicit none
complex, allocatable,dimension(:,:,:), pinned :: A , B
complex, allocatable,dimension(:,:,:), device :: A_d, B_d
real:: elapsed_time,scale
integer, parameter :: num_streams=8
integer:: nxx, nyy, nomega, plan, stream(num_streams)
integer :: clock_start,clock_end,clock_rate, istat, ifr, i, j, ind
logical:: plog

nxx=512;       nyy=512 ;   nomega=196

! Initialize FFT plan
call cufftPlan2d(plan,nyy,nxx,CUFFT_C2C)
do i = 1,num_streams
    istat= cudaStreamCreate(stream(i))
end do

! Find the clock rate
call SYSTEM_CLOCK(COUNT_RATE=clock_rate)
allocate(A(nxx,nyy,nomega),B(nxx,nyy,nomega))
allocate(A_d(nxx,nyy,num_streams),B_d(nxx,nyy,num_streams))

istat=cudaThreadSynchronize()
call SYSTEM_CLOCK(COUNT=clock_start) ! Start timing
scale =1./(nxx*nyy)
do ifr=1,nomega
    ind = mod(ifr,num_streams)+1

    ! Send data to GPU
    istat= cudaMemcpyAsync(A_d(1,1,ind),A(1,1,ifr),nxx*nyy, stream(ind))
    istat= cudaMemcpyAsync(B_d(1,1,ind),B(1,1,ifr),nxx*nyy, stream(ind))

    ! Execute FFTs on GPU
call cufftSetStream(plan,stream(ind))
call cufftExecC2C(plan ,A_d(1,1,ind),A_d(1,1,ind),CUFFT_FORWARD)
call cufftExecC2C(plan ,B_d(1,1,ind),B_d(1,1,ind),CUFFT_FORWARD)

    ! Convolution and scaling of the arrays
    !$cuf kernel do(2) <<<*,*,stream=stream(ind)>>>
do j=1,nyy
        do i=1,nxx
            B_d(i,j,ind)= A_d(i,j,ind)*B_d(i,j,ind)*scale
        end do
    end do

    ! Execute FFTs on GPU
call cufftExecC2C(plan ,B_d(1,1,ind),B_d(1,1,ind),CUFFT_INVERSE)
    ! Copy results back
    istat=cudaMemcpyAsync( B(1,1,ifr),B_d(1,1,ind),nxx*nyy, stream=stream(ind))
end do
istat=cudaThreadSynchronize()
call SYSTEM_CLOCK(COUNT=clock_end) ! End timing
print *,"Elapsed time :", REAL(clock_end-clock_start)/REAL(clock_rate)
deallocate(A,B,A_d,B_d)
end program
Computing $\pi$ with CUDA Fortran

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

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

- All the toolkit libraries have C interfaces
- Use F90 interfaces and ISO C Binding to use from CUDA Fortran

```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
end interface 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 $\pi$ 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
!
! Define how many numbers we want to generate
N=2000
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 -O3 -Mpreprocess -o pi_gpu precision_module.cuf curand_module.cuf pi.cuf -lcurand

<table>
<thead>
<tr>
<th>Samples</th>
<th>Pi</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>3.1112</td>
<td>0.3039E-01</td>
</tr>
<tr>
<td>100000</td>
<td>3.1363</td>
<td>0.5273E-02</td>
</tr>
<tr>
<td>1000000</td>
<td>3.1406</td>
<td>0.1029E-02</td>
</tr>
<tr>
<td>10000000</td>
<td>3.1409</td>
<td>0.6683E-03</td>
</tr>
<tr>
<td>100000000</td>
<td>3.1416</td>
<td>0.1192E-04</td>
</tr>
</tbody>
</table>
Computing $\pi$

```
pgf90 -O3 -Mpreprocess -o pi_gpu precision_module.cuf curand_module.cuf pi.cuf -lcurand
```

Where is the difference coming from?

```plaintext
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 ($x^2+y^2$), no special functions just + and *
Computing $\pi$

$\text{pgf90 -O3 -Mpreprocess -o pi\_gpu precision\_module.cuf curand\_module.cuf pi.cuf -lcurand}$

Where is the difference 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 ($x^2+y^2$), 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

<table>
<thead>
<tr>
<th>Samples</th>
<th>Pi=</th>
<th>Error=</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.16720009</td>
<td>0.2561E-01</td>
</tr>
<tr>
<td></td>
<td>3.13919997</td>
<td>0.2393E-02</td>
</tr>
<tr>
<td></td>
<td>3.14109206</td>
<td>0.5007E-03</td>
</tr>
<tr>
<td></td>
<td>3.14106607</td>
<td>0.5267E-03</td>
</tr>
<tr>
<td></td>
<td>3.14139462</td>
<td>0.1981E-03</td>
</tr>
</tbody>
</table>

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

- Parallelism across blocks
- Parallelism within a block
- No global synchronization
  - two-stage approach (two kernel launches), same code for both stages
## Parallel Reduction: Sequential Addressing

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Stride 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread IDs</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>Values</td>
<td>10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 2</th>
<th>Stride 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread IDs</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>Values</td>
<td>8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 3</th>
<th>Stride 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread IDs</td>
<td>1 2</td>
</tr>
<tr>
<td>Values</td>
<td>21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 4</th>
<th>Stride 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread IDs</td>
<td>1</td>
</tr>
<tr>
<td>Values</td>
<td>41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2</td>
</tr>
</tbody>
</table>
Computing $\pi$ with CUDA Fortran Kernels

```fortran
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)
```

82
Computing $\pi$ with CUDA Fortran Kernels

**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 $\pi$ with an 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 an 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)