The Abdus Salam
International Centre for Theoretical Physics

2068-24

Advanced School in High Performance and GRID Computing - Concepts and Applications

30 November - 11 December, 2009

Optimal GPU programming: Presentation and discussion of implementing a spectral Poisson solver with CUDA using the cuFFT library

M. Fatica
NVIDIA
U.S.A.

Strada Costiera 11, 34151 Trieste, Italy - Tel.+39 040 2240 111; Fax +39 040 224 163 - sci_info@ictp.it

NVIDIA®

CUDA Toolkit

CUDA

Driver: required component to run CUDA applications

Toolkit: compiler, CUBLAS and CUFFT
(required for development)

SDK: collection of examples and documentation

Support for Linux (32 and 64 bit), Windows XP and Vista (32 and 64 bit), MacOSX 10.5

Downloadable from http://www.nvidia.com/cuda

© NVIDIA Corporation 2008

CUDA Toolkit

Application Software

Industry Standard C Language

Libraries

cuFFT | cuBLAS | cuDPP

GPU:card, system

CUDA Compiler

C Fortran

CUDA Tools

Debugger Profiler

© NVIDIA Corporation 2008

CUDA Compiler: nvcc

- Any source file containing CUDA language extensions (.cu) must be compiled with nvcc
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC can output:
  - Either C code (CPU Code)
    - That must then be compiled with the rest of the application using another tool
  - Or PTX or object code directly
- An executable with CUDA code requires:
  - The CUDA core library (`cuda`)
  - The CUDA runtime library (`cudart`)

© NVIDIA Corporation 2008

CUDA Compiler: nvcc

- Important flags:
  - -arch sm_13 — Enable double precision (on compatible hardware)
  - -G — Enable debug for device code
  - --ptxas-options=-v — Show register and memory usage
  - --maxrregcount <N> — Limit the number of registers
  - -use_fast_math — Use fast math library

© NVIDIA Corporation 2008

CUDA libraries

- CUDA includes 2 widely used libraries
  - CUBLAS: BLAS implementation
  - CUFFT: FFT implementation

- CUDPP (Data Parallel Primitives), available from http://www.gpgpu.org/developer/cudpp/ :
  - Reduction
  - Scan
  - Sort

© NVIDIA Corporation 2008

CUFFT

- The Fast Fourier Transform (FFT) is a divide-and-conquer algorithm for efficiently computing discrete Fourier transform of complex or real-valued data sets.
- CUFFT is the CUDA FFT library
  - Provides a simple interface for computing parallel FFT on an NVIDIA GPU
  - Allows users to leverage the floating-point power and parallelism of the GPU without having to develop a custom, GPU-based FFT implementation

© NVIDIA Corporation 2008

Supported Features

- 1D, 2D and 3D transforms of complex and real-valued data
- Batched execution for doing multiple 1D transforms in parallel
- 1D transform size up to 8M elements
- 2D and 3D transform sizes in the range [2,16384]
- In-place and out-of-place transforms for real and complex data.

© NVIDIA Corporation 2008

Transform Types

- Library supports real and complex transforms
  - CUFFT_C2C, CUFFT_C2R, CUFFT_R2C
- Directions
  - CUFFT_FORWARD (-1) and CUFFT_INVERSE (1)
    - According to sign of the complex exponential term
- Real and imaginary parts of complex input and output arrays are interleaved
  - cufftComplex type is defined for this
- Real to complex FFTs, output array holds only nonredundant coefficients
  - N -> N/2+1
  - N0 x N1 x … x Nn -> N0 x N1 x … x (Nn/2+1)
  - For in-place transforms the input/output arrays need to be padded

© NVIDIA Corporation 2008

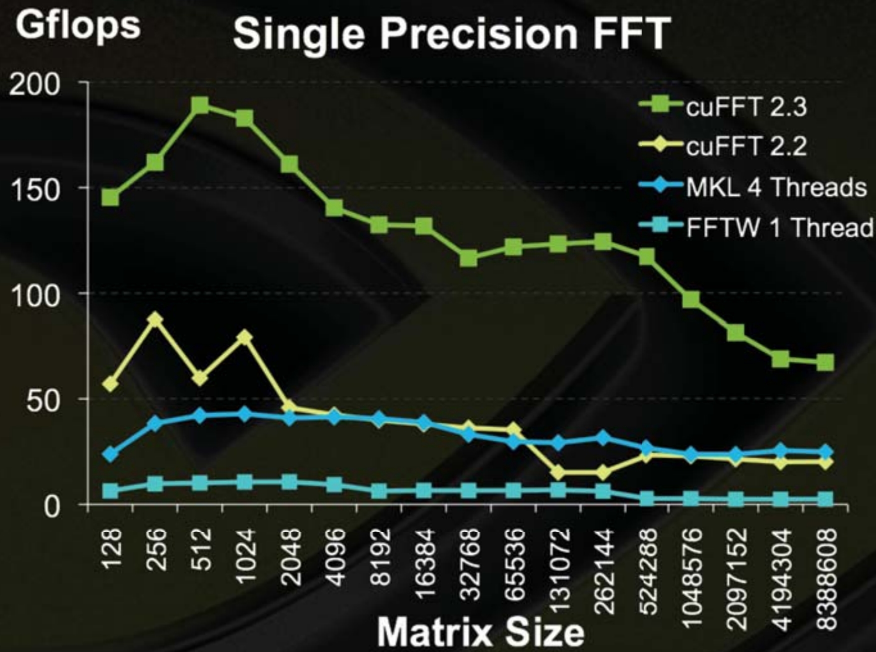
More on Transforms

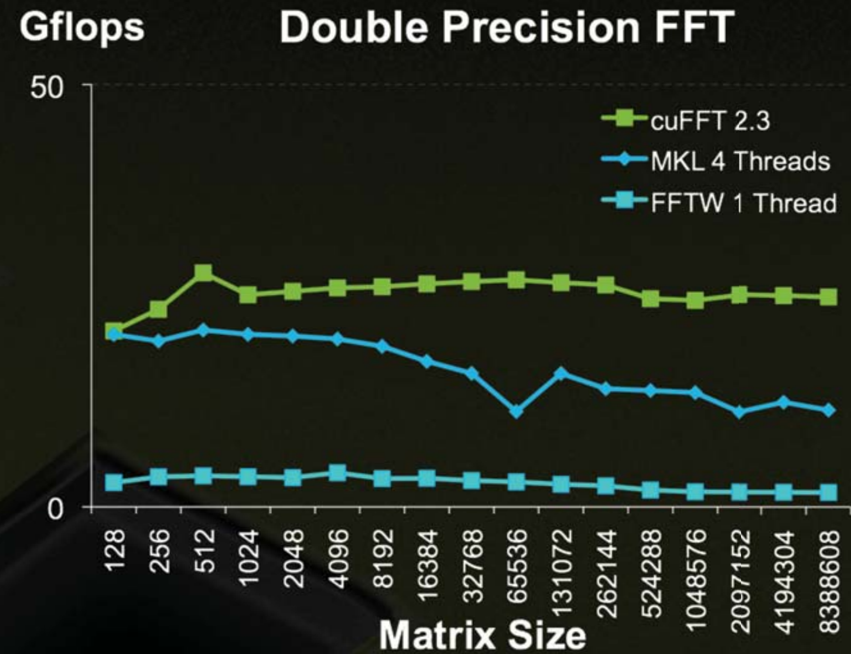
- For 2D and 3D transforms, CUFFT performs transforms in row-major (C-order)
- If calling from FORTRAN or MATLAB, remember to change the order of size parameters during plan creation
- CUFFT performs un-normalized transforms:

  IFFT(FFT(A))= length(A)*A
- CUFFT API is modeled after FFTW. Based on plans, that completely specify the optimal configuration to execute a particular size of FFT
- Once a plan is created, the library stores whatever state is needed to execute the plan multiple times without recomputing the configuration
  - Works very well for CUFFT, because different kinds of FFTs require different thread configurations and GPU resources

© NVIDIA Corporation 2008

FFT Performance: CPU vs GPU





cuFFT 2.3 beta: NVIDIA Tesla C1060 GPU

MKL 10.1r1: Quad-Core Intel Core i7 (Nehalem) 3.2GHz

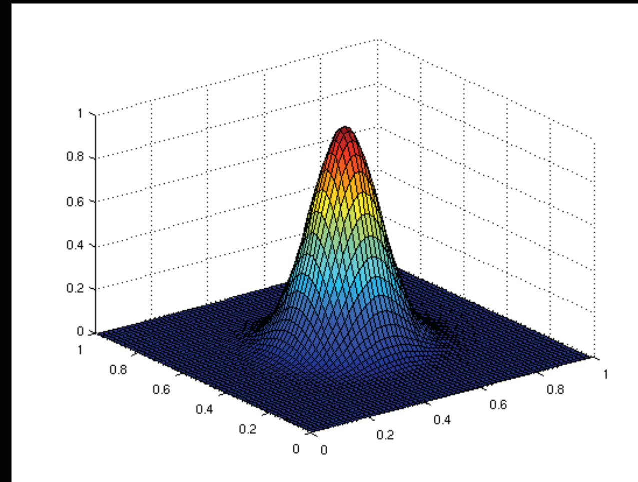
Code example:
2D complex to complex transform

```
#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. */
 cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);

/* Note:
    Different pointers to input and output arrays implies out of place transformation
 */

 /* Destroy the CUFFT plan. */
   cufftDestroy(plan);

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

© NVIDIA Corporation 2008

CUDA Example:
Fourier-spectral Poisson Solver

Solve a Poisson equation on a rectangular domain with periodic boundary conditions using a Fourier-spectral method.



This example will show how to use the FFT library, transfer the data to/from GPU and perform simple computations on the GPU.

© NVIDIA Corporation 2008

Mathematical background

$$\nabla^2 \phi = r \xrightarrow{FFT} -(k_x^2 + k_y^2)\hat{\phi} = \hat{r}$$

1. Apply 2D forward FFT to r to obtain r(k), where k is the wave number

2. Apply the inverse of the Laplace operator to r(k) to obtain u(k): simple element-wise division in Fourier space

$$\hat{\phi} = -\frac{\hat{r}}{(k_x^2 + k_y^2)}$$

3. Apply 2D inverse FFT to u(k) to obtain u

© NVIDIA Corporation 2008

Reference MATLAB implementation

```
% No. of Fourier modes
      N = 64;
% Domain size (assumed square)
      L = 1;
% Characteristic width of f (make << 1)
      sig = 0.1;
% Vector of wavenumbers
      k = (2*pi/L)*[0:(N/2-1) (-N/2):(-1)];
%Matrix of (x,y) wavenumbers corresponding
% to Fourier mode (m,n)
      [KX KY]  = meshgrid(k,k);
% Laplacian matrix acting on the wavenumbers
      delsq = -(KX.^2 + KY.^2);
% Kludge to avoid division by zero for
% wavenumber (0,0).
% (this waveno. of fhat  should be zero anyway!)
      delsq(1,1) = 1;
% Grid spacing
      h = L/N;
      x = (0:(N-1))*h ;
      y = (0:(N-1))*h;
      [X Y] = meshgrid(x,y);
% Construct RHS f(x,y) at the Fourier gridpoints
      rsq = (X-0.5*L).^2 + (Y-0.5*L).^2;
      sigsq = sig^2;
      f = exp(-rsq/(2*sigsq)).*…
          (rsq - 2*sigsq)/(sigsq^2);
% Spectral inversion of Laplacian
      fhat = fft2(f);
      u = real(ifft2(fhat./delsq));
% Specify arbitrary constant by forcing corner
%  u = 0.
      u = u - u(1,1);
%  Compute L2 and Linf norm of error
      uex = exp(-rsq/(2*sigsq));
      errmax = norm(u(:)-uex(:),inf);
      errmax2 = norm(u(:)-uex(:),2)/(N*N);
%  Print L2 and Linf norm of error
      fprintf('N=%d\n',N);
      fprintf('Solution at (%d,%d): ',N/2,N/2);
      fprintf('computed=%10.6f …
          reference = %10.6f\n',u(N/2,N/2),
      uex(N/2,N/2));
      fprintf('Linf err=%10.6e L2 norm
              err = %10.6e\n',errmax, errmax2);
```

http://www.atmos.washington.edu/2005Q2/581/matlab/pois_FFT.m

© NVIDIA Corporation 2008

Implementation steps

The following steps need to be performed:

1. Allocate memory on host: r (NxN), u (NxN) , kx (N) and ky (N)
2. Allocate memory on device: r_d, u_d, kx_d, ky_d
3. Transfer r, kx and ky from host memory to the correspondent arrays on device memory
4. Initialize plan for FFT
5. Compute execution configuration
6. Transform real input to complex input
7. 2D forward FFT
8. Solve Poisson equation in Fourier space
9. 2D inverse FFT
10. Transform complex output to real input and apply scaling
11. Transfer results from the GPU back to the host

We are not taking advantage of the symmetries (C2C transform for real data) to keep the code simple.

© NVIDIA Corporation 2008

Solution walk-through (steps 1-2)

```
/*Allocate arrays on the host */
    float *kx, *ky, *r;
    kx = (float *) malloc(sizeof(float*N);
    ky = (float *) malloc(sizeof(float*N);
    r  = (float *) malloc(sizeof(float*N*N);

/* Allocate array on the GPU with cudaMalloc */
    float *kx_d, *ky_d, *r_d;
    cudaMalloc( (void **) &kx_d, sizeof(cufftComplex)*N);
    cudaMalloc( (void **) &ky_d, sizeof(cufftComplex)*N);
    cudaMalloc( (void **) &r_d  , sizeof(cufftComplex)*N*N);

    cufftComplex *r_complex_d;
    cudaMalloc( (void **) &r_complex_d, sizeof(cufftComplex)*N*N);
```

© NVIDIA Corporation 2008

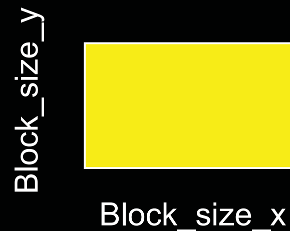
Code walk-through (steps 3-4)

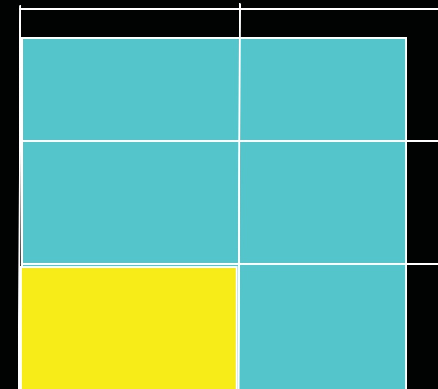
```
/* Initialize r, kx and ky on the host */
   ...............
/*Transfer data from host to device with
   cudaMemcpy(target, source, size, direction)*/
  cudaMemcpy (kx_d, kx, sizeof(float)*N    , cudaMemcpyHostToDevice);
  cudaMemcpy (ky_d, ky, sizeof(float)*N    , cudaMemcpyHostToDevice);
  cudaMemcpy (r_d  , r   , sizeof(float)*N*N, cudaMemcpyHostToDevice);

/* Create plan for CUDA FFT (interface similar to FFTW) */
   cufftHandle plan;
   cufftPlan2d( &plan, N, N, CUFFT_C2C);
```

© NVIDIA Corporation 2008

Code walk-through (step 5)

```
/* Compute the execution configuration
    NB: block_size_x*block_size_y = number of threads
          On G80  number of threads < 512  */
  dim3 dimBlock(block_size_x, block_size_y);
  dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);

/* Handle N not multiple of block_size_x or block_size_y */
  if (N % block_size_x  !=0 ) dimGrid.x+=1;
  if (N % block_size_y  !=0 ) dimGrid.y+=1
```





© NVIDIA Corporation 2008

Code walk-through (step 6-10)

```
/* Transform real input to complex input */
   real2complex<<<dimGrid, dimBlock>>> (r_d, r_complex_d, N);

/* Compute in place forward FFT */
   cufftExecC2C (plan, r_complex_d, r_complex_d, CUFFT_FORWARD);

/* Solve Poisson equation in Fourier space */
   solve_poisson<<<dimGrid, dimBlock>>> (r_complex_d, kx_d, ky_d,N);

/* Compute in place inverse FFT */
   cufftExecC2C (plan, r_complex_d, r_complex_d, CUFFT_INVERSE);

 /* Copy the solution back to a real array and apply scaling ( an FFT followed by iFFT will
      give you back the same array times the length of the transform) */
   scale = 1.f / ( (float) N * (float) N );
   complex2real_scaled<<<dimGrid, dimBlock>>> (r_d, r_complex_d, N, scale);
```

© NVIDIA Corporation 2008

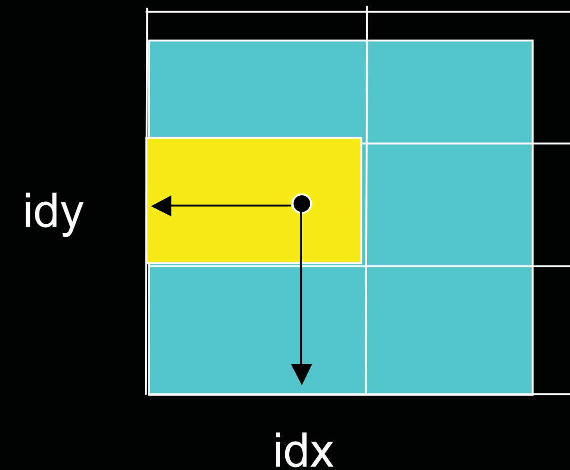
Code walk-through (step 11)

```
/*Transfer data from device to host  with
   cudaMemcpy(target, source, size, direction)*/
  cudaMemcpy (r   , r_d    , sizeof(float)*N*N, cudaMemcpyDeviceToHost);

/* Destroy  plan and clean up memory on device*/
   cufftDestroy( plan);
   cudaFree(r_complex_d);
   …….
   cudaFree(kx_d);
```

© NVIDIA Corporation 2008

real2complex

```
/*Copy real data to complex data */

 __global__ void real2complex (float *a,  cufftComplex *c, int N)
{
       /* compute idx and idy, the location of the element in the original NxN array */
       int idx = blockIdx.x*blockDim.x+threadIdx.x;
       int idy = blockIdx.y*blockDim.y+threadIdx.y;

       if ( idx < N && idy <N)
              {
               int index = idx + idy*N;
               c[index].x = a[index];
               c[index].y = 0.f;
              }
}
```



© NVIDIA Corporation 2008

solve_poisson

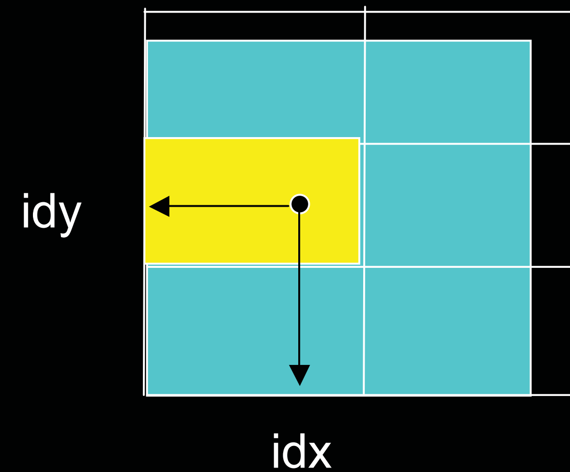
```
__global__ void solve_poisson (cufftComplex *c, float *kx, float *ky, int N)
{
        /* compute idx and idy, the location of the element in the original NxN array */
        int idx = blockId.x*blockDim.x+threadIdx.x;
        int idy = blockId.y*blockDim.y+threadIdx.y;

        if ( idx < N && idy <N)
                {
                 int index = idx + idy*N;
                 float scale =  -  ( kx[idx]*kx[idx] + ky[idy]*ky[idy] );
                 if ( idx ==0 && idy == 0 ) scale =1.f;
                 scale = 1.f / scale;
                 c[index].x *= scale;
                 c[index].y *= scale;
                }
}
```

$$\hat{\phi} = -\frac{\hat{r}}{(k_x^2 + k_y^2)}$$

© NVIDIA Corporation 2008

complex2real_scaled

```
/*Copy real part of complex data into real array and apply scaling */

 __global__ void complex2real_scaled (cufftComplex *c, float *a, int N,
                                                       float scale)
{
      /* compute idx and idy, the location of the element in the original NxN array */
      int idx = blockId.x*blockDim.x+threadIdx.x;
      int idy = blockId.y*blockDim.y+threadIdx.y;

      if ( idx < N && idy <N)
            {
             int index = idx + idy*N;
             a[index] = scale*c[index].x ;
             }
}
```



© NVIDIA Corporation 2008

Compile and run poisson

- **Compile the example poisson.cu:**

```
nvcc –O3 –o poisson poisson.cu
-I/usr/local/cuda/include –L/usr/local/cuda/lib  -lcufft
-L/usr/local/NVDIA_CUDA_SDK/common/inc
-L/usr/local/NVDIA_CUDA_SDK/lib -lcutil
```

- **Run the example**

```
./poisson -N64
Poisson solver on a domain  64 x 64
dimBlock 32 16 (512 threads)
dimGrid  2 4
L2 error 9.436995e-08:
Time       0.000569:
Time I/O 0.000200 (0.000136 + 0.000064):
Solution at (32,32)
computed=0.975879 reference=0.975882
```

- **Reference values from MATLAB:**

```
N=64
Solution at (32,32): computed=  0.975879 reference=   0.975882
Linf err=2.404194e-05 L2 norm err = 9.412790e-08
```

© NVIDIA Corporation 2008

Profiling

Profiling the function calls in CUDA is very easy.

It is controlled via environment variables:

- CUDA_PROFILE: to enable or disable
  - 1 (enable profiler)
  - 0 (default, no profiler)
- CUDA_PROFILE_LOG: to specify the filename
  - If set, it will write to “filename”
  - If not set, it will write to cuda_profile.log
- CUDA_PROFILE_CSV: control the format
  - 1 (enable comma separated file)
  - 0 (disable comma separated file)

© NVIDIA Corporation 2008

Profiler output from Poisson_1

./poisson_1 –N1024

```
method=[ memcopy ] gputime=[ 1427.200 ]
method=[ memcopy ] gputime=[ 10.112 ]
method=[ memcopy ] gputime=[ 9.632 ]

method=[ real2complex ] gputime=[ 1654.080 ] cputime=[ 1702.000 ] occupancy=[ 0.667 ]

method=[ c2c_radix4 ]     gputime=[ 8651.936 ] cputime=[ 8683.000 ] occupancy=[ 0.333 ]
method=[ transpose ]      gputime=[ 2728.640 ] cputime=[ 2773.000 ] occupancy=[ 0.333 ]
method=[ c2c_radix4 ]     gputime=[ 8619.968 ] cputime=[ 8651.000 ] occupancy=[ 0.333 ]
method=[ c2c_transpose ] gputime=[ 2731.456 ] cputime=[ 2762.000 ] occupancy=[ 0.333 ]

method=[ solve_poisson] gputime=[ 6389.984 ] cputime=[ 6422.000 ] occupancy=[ 0.667 ]

method=[ c2c_radix4 ] gputime=[ 8518.208 ] cputime=[ 8556.000 ] occupancy=[ 0.333 ]
method=[ c2c_transpose] gputime=[ 2724.000 ] cputime=[ 2757.000 ] occupancy=[ 0.333 ]
method=[ c2c_radix4 ] gputime=[ 8618.752 ] cputime=[ 8652.000 ] occupancy=[ 0.333 ]
method=[ c2c_transpose] gputime=[ 2767.840 ] cputime=[ 5248.000 ] occupancy=[ 0.333 ]

method=[ complex2real_scaled ] gputime=[ 2844.096 ] cputime=[ 3613.000 ] occupancy=[ 0.667 ]

method=[ memcopy ] gputime=[ 2461.312 ]
```

© NVIDIA Corporation 2008

Improving performances

- **Use pinned memory to improve CPU/GPU transfer time:**

```
#ifdef PINNED
        cudaMallocHost((void **) &r,sizeof(float)*N*N); // rhs, 2D array
#else
        r  = (float *) malloc(sizeof(float)*N*N); // rhs, 2D array
#endif
```

```
$ ./poisson_1 -N1024
dimBlock 32 16 (512 threads)  dimGrid  32 64   L2 error 5.663106e-09:

Total Time    : 6.533000 (ms)
Solution Time: 2.938000 (ms)
Time I/O      : 3.468000 (1.023000 + 2.445000) (ms)

Solution at (512,512)  computed=0.999902 reference=0.999905

$ ./poisson_1_pinned -N1024
dimBlock 32 16 (512 threads) dimGrid  32 64 L2 error 5.663106e-09:

Total Time    : 4.686000 (ms)
Solution Time: 2.945000 (ms)
Time I/O      : 1.644000 (0.886000 + 0.758000) (ms)

Solution at (512,512) computed=0.999902 reference=0.999905
```

© NVIDIA Corporation 2008

Additional improvements

- **Use shared memory for the arrays kx and ky in solve_poisson**
- **Use fast integer operations (__umul24)**

© NVIDIA Corporation 2008

solve_poisson (with shared memory)

```
__global__ void solve_poisson (cufftComplex *c, float *kx, float *ky, int N)
{
        unsigned int idx   = __umul24(blockIdx.x,blockDim.x)+threadIdx.x;
        unsigned int idy   = __umul24(blockIdx.y,blockDim.y)+threadIdx.y;
        // use shared memory to minimize multiple access to same k values
      __shared__ float kx_s[BLOCK_WIDTH], ky_s[BLOCK_HEIGHT]
       if (threadIx.x < 1) kx_s[threadIdx.x] = kx[idx];
       if (threadIx.y < 1) ky_s[threadIdx.y] = ky[idy];
       __syncthreads();
        if ( idx < N && idy <N)
                {
                unsigned int index = idx +__umul24(idy ,N);
                float scale = -  ( kx_s[threadIdx.x]*kx_s[threadIdx.x]
                                       + ky_s[threadIdy.y]*ky_s[threadIdy.y] );
                if ( idx ==0 && idy == 0 ) scale =1.f;
                scale = 1.f / scale;
                 c[index].x *= scale;
                 c[index].y*= scale;
                }
}
```

$$\hat{\phi} = -\frac{\hat{r}}{(k_x^2 + k_y^2)}$$

© NVIDIA Corporation 2008

Profiler output from Poisson_2

./poisson_2 –N1024 –x16 –y16

method=[memcopy] gputime=[1426.048]
method=[memcopy] gputime=[9.760]
method=[memcopy] gputime=[9.472]

method=[real2complex] gputime=[1611.616] cputime=[1662.000] occupancy=[0.667] (was 1654)

method=[c2c_radix4] gputime=[8658.304] cputime=[8689.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2731.424] cputime=[2763.000] occupancy=[0.333]
method=[c2c_radix4] gputime=[8622.048] cputime=[8652.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2738.592] cputime=[2770.000] occupancy=[0.333]

method=[solve_poisson] gputime=[2760.192] cputime=[2792.000] occupancy=[0.667] (was 6389)

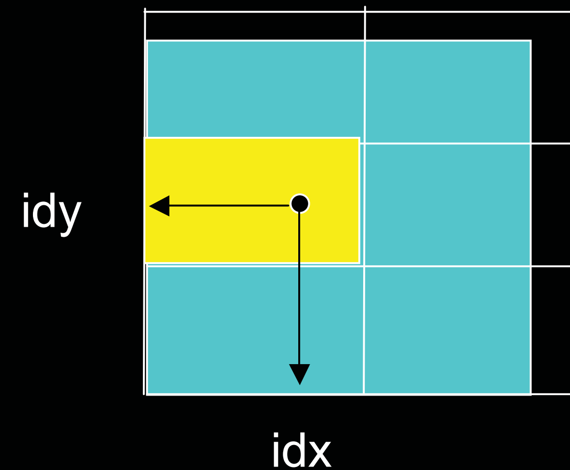
method=[c2c_radix4] gputime=[8517.952] cputime=[8550.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2729.632] cputime=[2766.000] occupancy=[0.333]
method=[c2c_radix4] gputime=[8621.024] cputime=[8653.000] occupancy=[0.333]
method=[c2c_transpose] gputime=[2770.912] cputime=[5252.000] occupancy=[0.333]

method=[complex2real_scaled] gputime=[2847.008] cputime=[3616.000] occupancy=[0.667]
method=[memcopy] gputime=[2459.872]

© NVIDIA Corporation 2008

complex2real_scaled (fast version)

```
__global__ void complex2real_scaled (cufftComplex *c, float *a, int N, float
    scale)
{
    /* compute idx and idy, the location of the element in the original NxN array */
    int idx = blockId.x*blockDim.x+threadIdx.x;
    int idy = blockId.y*blockDim.y+threadIdx.y;
    volatile float2 c2;
        if ( idx < N && idy <N)
        {
         int index = idx + idy*N;
         c2.x= c[index].x;
         c2.y= c[index].y;
         a[index] = scale*c2.x ;
         }
}
```



From the ptx file, we discover that the compiler is optimizing out the vector load which prevents memory coalescing. Use volatile to force vector load

© NVIDIA Corporation 2008

Profiler output from Poisson_3

```
method=[ memcopy ] gputime=[ 1427.808 ]
method=[ memcopy ] gputime=[ 9.856 ]
method=[ memcopy ] gputime=[ 9.600 ]

method=[ real2complex] gputime=[ 1614.144 ] cputime=[ 1662.000 ] occupancy=[ 0.667 ]

method=[ c2c_radix4]       gputime=[ 8656.800 ] cputime=[ 8688.000 ] occupancy=[ 0.333 ]
method=[ c2c_transpose] gputime=[ 2727.200 ] cputime=[ 2758.000 ] occupancy=[ 0.333 ]
method=[ c2c_radix4 ]      gputime=[ 8607.616 ] cputime=[ 8638.000 ] occupancy=[ 0.333 ]
method=[ c2c_transpose] gputime=[ 2729.888 ] cputime=[ 2761.000 ] occupancy=[ 0.333 ]

method=[ solve_poisson ] gputime=[ 2762.656 ] cputime=[ 2794.000 ] occupancy=[ 0.667 ]

method=[ c2c_radix4 ]      gputime=[ 8514.720 ] cputime=[ 8547.000 ] occupancy=[ 0.333 ]
method=[ c2c_transpose] gputime=[ 2724.192 ] cputime=[ 2760.000 ] occupancy=[ 0.333 ]
method=[ c2c_radix4 ]     gputime=[ 8620.064 ] cputime=[ 8652.000 ] occupancy=[ 0.333 ]
method=[ c2c_transpose] gputime=[ 2773.920 ] cputime=[ 4270.000 ] occupancy=[ 0.333 ]

method=[ complex2real_scaled ] gputime=[ 1524.992 ] cputime=[ 1562.000 ] occupancy=[ 0.667 ]

method=[ memcopy ] gputime=[ 2468.288 ]
```

© NVIDIA Corporation 2008

Performance improvement

	Non-pinned memory	Pinned memory
Initial implementation (r2c, poisson, c2r)	67ms (10.8ms)	63ms
+Shared memory +Fast integer mul	63.4ms (7.1ms)	59.4ms
+Coalesced read in c2r	62.1ms (5.8ms)	58.2ms

Tesla C870, pinned memory, optimized version: 10.4ms
Tesla C1060, pinned memory, optimized version: 4.65ms

© NVIDIA Corporation 2008

CUBLAS

- Implementation of BLAS (Basic Linear Algebra Subprograms) on top of CUDA driver
  - Self-contained at the API level, no direct interaction with CUDA driver
- Basic model for use
  - Create matrix and vector objects in GPU memory space
  - Fill objects with data
  - Call sequence of CUBLAS functions
  - Retrieve data from GPU
- CUBLAS library contains helper functions
  - Creating and destroying objects in GPU space
  - Writing data to and retrieving data from objects

© NVIDIA Corporation 2008

Supported Features

- BLAS functions
  - Single precision data:
    - Level 1 (vector-vector O(N))
    - Level 2 (matrix-vector $O(N^2)$)
    - Level 3 (matrix-matrix $O(N^3)$)
  - Complex single precision data:
    - Level 1
    - CGEMM
  - Double precision data:
    - Level 1: DASUM, DAXPY, DCOPY, DDOT, DNRM2, DROT, DROTM, DSCAL, DSWAP, ISAMAX, IDAMIN
    - Level 2: DGEMV, DGER, DSYR, DTRSV
    - Level 3: ZGEMM, DGEMM, DTRSM, DTRMM, DSYMM, DSYRK, DSYR2K
- Following BLAS convention, CUBLAS uses column-major storage

© NVIDIA Corporation 2008

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
- Implemented using C-based CUDA tool chain
  - Interfacing to C/C++ applications is trivial

© NVIDIA Corporation 2008

Calling CUBLAS from FORTRAN

- **Two interfaces:**
  - **Thunking** (define CUBLAS_USE_THUNKING when compiling fortran.c)
    - 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)

© NVIDIA Corporation 2008

SGEMM example (THUNKING)

```
! Define  3  single precision matrices A, B, C
real , dimension(m1,m1)::     A, B, C
……
! Initialize
……
#ifdef CUBLAS
 ! Call SGEMM in CUBLAS library using THUNKING interface (library takes care of
 ! memory allocation on device and data movement)
  call cublasSGEMM ('n','n',m1,m1,m1,alpha,A,m1,B,m1,beta,C,m1)
#else
! Call SGEMM in  host BLAS library
  call SGEMM ('n','n',m1,m1,m1,alpha,A,m1,B,m1,beta,C,m1)
#endif
```

To use the host BLAS routine:

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

To use the CUBLAS routine (fortran.c is provided by NVIDIA):

gcc -O3 -DCUBLAS_USE_THUNKING -I/usr/local/cuda/include -c fortran.c

g95 -O3 -DCUBLAS code.f90 fortran.o -L/usr/local/cuda/lib -lcublas

© NVIDIA Corporation 2008

SGEMM example (NON-THUNKING)

```
! Define  3  single precision matrices A, B, C
  real , dimension(m1,m1)::      A, B, C
  integer::  devPtrA, devPtrB, devPtrC, size_of_real=4
……
! Initialize A, B, C
………
! Allocate matrices on GPU
  cublasAlloc(m1*m1, size_of_real, devPtrA)
  cublasAlloc(m1*m1, size_of_real, devPtrB)
  cublasAlloc(m1*m1, size_of_real, devPtrC)
!Copy data from CPU to GPU
  cublasSetMatrix(m1,m1, size_of_real, A,m1, devPtrA, m1)
  cublasSetMatrix(m1,m1, size_of_real, B,m1, devPtrB, m1)
  cublasSetMatrix(m1,m1, size_of_real, C,m1, devPtrC, m1)
! Call SGEMM in CUBLAS library using NON-THUNKING interface (library is expecting data in GPU memory)
  call cublasSGEMM ('n','n',m1,m1,m1,alpha,devPtrA,m1,devPtrB,m1,beta,devPtrC,m1)
!Copy data from GPU to CPU
 cublasGetMatrix(m1,m1, size_of_real, devPtrC,m1, C, m1)
! Free memory on device
  cublasFree(devPtrA)
……
```

```
g95 -O3 code.f90  -L/usr/local/cuda/lib  -lcublas
```

© NVIDIA Corporation 2008

DGEMM Performance

GFLOPs

120
100
80
60
40
20
0

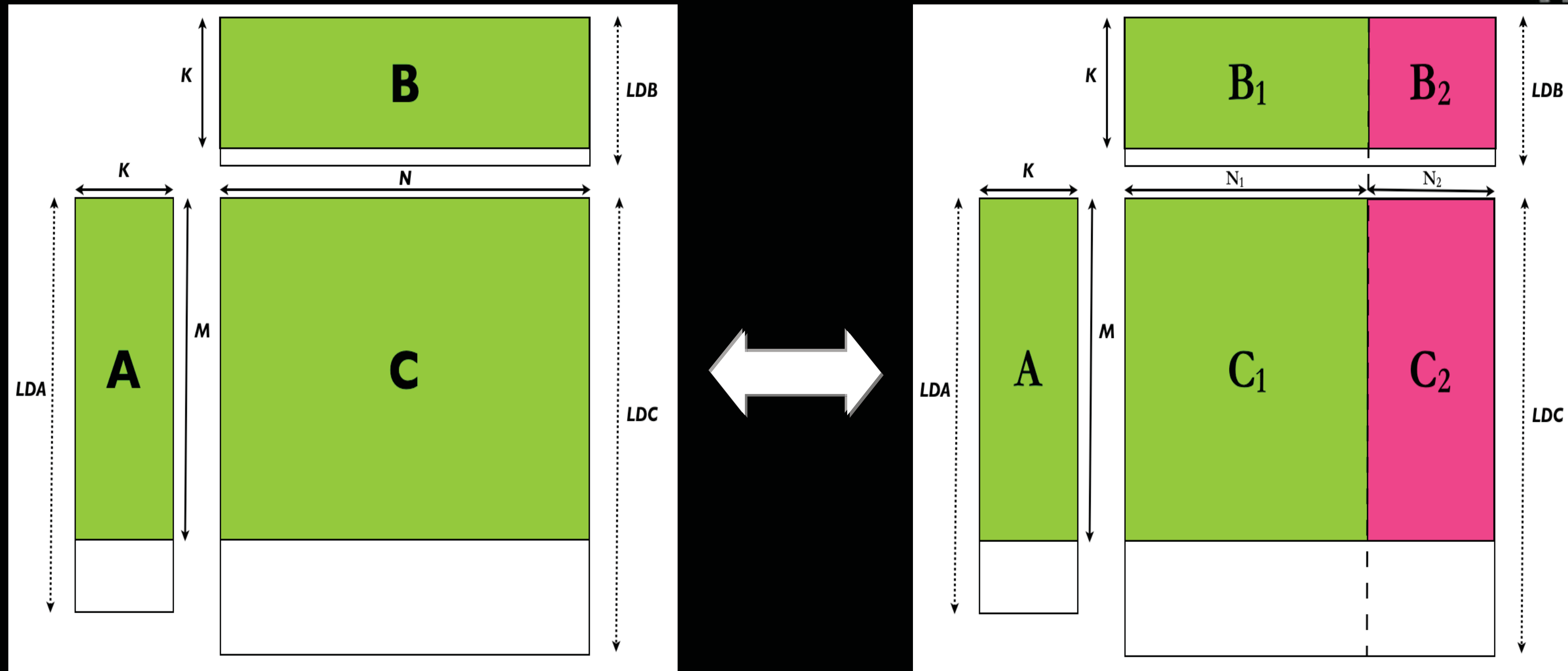
128 384 640 896 1152 1408 1664 1920 2176 2432 2688 2944 3200 3456 3712 3968 4224 4480 4736 4992 5248 5504 5760 6016

Size

Xeon Quad-core 2.8 GHz, MKL 10.3
Tesla C1060 GPU (1.296 GHz)
CPU + GPU

© NVIDIA Corporation 2008

DGEMM: C = alpha A B + beta C



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

(GPU) (CPU)

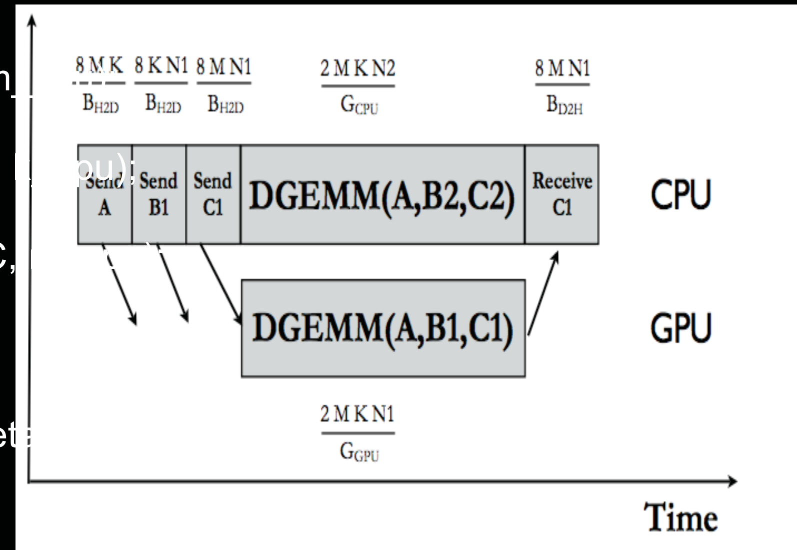
The idea can be extended to multi-GPU configuration and to handle huge matrices

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

© NVIDIA Corporation 2008

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_
// Copy B1  from CPU memory to GPU memory devB
   status = cublasSetMatrix (k ,n_gpu, sizeof(B[0]), B, ldb, devB, 
// Copy C1  from CPU memory to GPU memory devC
   status = cublasSetMatrix (m, n_gpu, sizeof(C[0]), C, ldc, devC,

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

// 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

© NVIDIA Corporation 2008

DGEMM performance on GPU

A DGEMM call in CUBLAS maps to several different kernels depending on the size c
With the combined CPU/GPU approach, we can always send optimal work to the GPU.

M	K	N	M%64	K%16	N%16	Gflop s
448	400	12320	Y	Y	Y	82.4
12320	400	1600	N	Y	Y	75.2
12320	300	448	N	N	Y	55.9
12320	300	300	N	N	N	55.9

Tesla T10 1.44Ghz, data resident in GPU memory. Optimal kernel achieves 95% o

© NVIDIA Corporation 2008

Optimal split

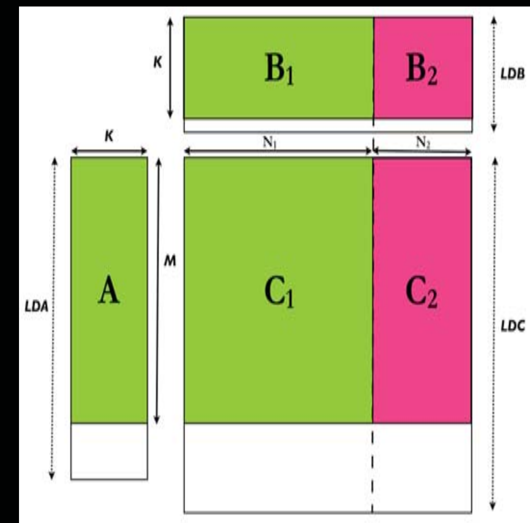
If A(M,K), B(K,N) and C(M,N), a DGEMM call performs 2*M*K*N operations

$T_{CPU}(M,K,N2) = T_{GPU}(M,k,N1) \qquad N=N1+N2$

If G_{CPU} denotes the DGEMM performance of the CPU in Gflops and G_{GPU} the one of the GPU,

The optimal split is

$\eta = G_{GPU} / (G_{CPU}+G_{GPU})$



© NVIDIA Corporation 2008

Interfacing CUDA with other languages

- CUDA kernels from FORTRAN, allocate pinned memory from FORTRAN
- Calling CUDA from MATLAB with MEX files
- Several packages (open source and commercial) to interface CUDA with Python, IDL, .NET, FORTRAN (Flagon). Browse CUDA Zone to find all the packages.

© NVIDIA Corporation 2008

Pinned memory from FORTRAN

Pinned memory provides a fast PCI-e transfer speed and enables use of streams:

•Allocation needs to be done with cudaMallocHost

•Use new Fortran 2003 features for interoperability with C.

```
use iso_c_binding
! The allocation is performed by C function calls. Define the C pointer as type (C_PTR)
 type(C_PTR) :: cptr_A, cptr_B, cptr_C
! Define Fortran arrays as pointer.
real, dimension(:,:), pointer ::      A, B, C

! Allocating memory with cudaMallocHost.
! The Fortan arrays, now defined as pointers, are then associated with the C pointers using the
! new interoperability defined in iso_c_binding. This is equivalent to allocate(A(m1,m1))
 res = cudaMallocHost ( cptr_A, m1*m1*sizeof(fp_kind) )
 call c_f_pointer ( cptr_A, A, (/ m1, m1 /) )

! Use A as usual.
! See example code for cudaMallocHost interface code
```

http://www.nvidia.com/object/cuda_programming_tools.html

© NVIDIA Corporation 2008

Calling CUDA kernels from FORTRAN

From Fortran call C function that will call CUDA kernel

```
! Fortran -> C -> CUDA ->C ->Fortran
call cudafunction(c,c2,N)
```

```
/* NB: Fortran subroutine arguments are passed by reference.    */
extern "C" void cudafunction_(cuComplex *a, cuComplex *b,  int *Np)
{
  ...
  int N=*np;
  cudaMalloc ((void **) &a_d , sizeof(cuComplex)*N);
  cudaMemcpy( a_d, a,  sizeof(cuComplex)*N   ,cudaMemcpyHostToDevice);
  dim3 dimBlock(block_size); dim3 dimGrid (N/dimBlock.x); if( N % block_size != 0 ) dimGrid.x+=1;
  square_complex<<<dimGrid,dimBlock>>>(a_d,a_d,N);
   cudaMemcpy( b, a_d, sizeof(cuComplex)*N,cudaMemcpyDeviceToHost);
  cudaFree(a_d);
}
```

```
complex_mul: main.f90 Cuda_function.o
	$(FC) -o complex_mul main.f90 Cuda_function.o -L/usr/local/cuda/lib  -lcudart -lstdc++

cuda_function.o: cuda_function.cu
	nvcc -c -O3 cuda_function.cu
```

© NVIDIA Corporation 2008

Fortran for CUDA

- Collaborative NVIDIA/PGI effort
- Part of PGI pgf90 compiler
  - Program host and device code similar to CUDA C
  - Host code based on Runtime API
  - Separate from PGI Accelerator
    - Directive-based, OpenMP-like interface to CUDA
- Timeline
  - Currently device emulation
  - Alpha in September
  - Release by SC09

© NVIDIA Corporation 2009
© NVIDIA Corporation 2008

Array Increment Example

```
program increment

use cudafor
implicit none

integer, parameter :: n = 10000
integer, parameter :: blocksize = 256
type (dim3) :: dimGrid, dimBlock

real, allocatable :: h_data(:)
real, device, allocatable :: d_data(:)

! host allocation and initialization
allocate(h_data(n))
h_data = 1

! device allocation
allocate(d_data(n))

! Host to device transfer
d_data = h_data
```

© NVIDIA Corporation 2008

Array Increment Example

```
! execution configuration parameters
  dimGrid = dim3(ceiling(real(n)/blocksize), 1, 1)
  dimBlock = dim3(blocksize, 1, 1)

  ! launch kernel
  call increment_gpu<<<dimGrid, dimBlock>>>(d_data, n)
  ! device to host transfer
  h_data = d_data
  ...
  deallocate(h_data, d_data)

contains

  attributes(global) subroutine increment_gpu(a, nele)
    real, intent(inout) :: a(*)
    integer, intent(in), value :: nele
    integer :: idx

    idx = (blockidx%x-1)*blockdim%x + threadidx%x  ! unit offset
    if (idx <= nele) a(idx) = a(idx)+1
  end subroutine increment_gpu

end program increment
```

© NVIDIA Corporation 2008

Availability and Additional Information

- CUDA Fortran will be supported on CUDA-enabled NVIDIA GPUs in the PGI 10.0 Fortran 95/03 compiler scheduled to be available in November, 2009
- PGI CUDA Fortran will be supported on Linux, MacOS and Windows
- PGI CUDA Fortran will include support for a host emulation mode for debugging
- See http://www.pgroup.com/resources/cudafortran.htm for a detailed specification of CUDA Fortran

© NVIDIA Corporation 2008

CUDA & MATLAB

- Even though MATLAB is built on many well-optimized libraries, some functions can perform better when written in a compiled language (e.g. C and Fortran).

- MATLAB provides a convenient API for interfacing code written in C and FORTRAN to MATLAB functions with MEX files.

- MEX files could be used to exploit multi-core processors with OpenMP or threaded codes or like in this case to offload functions to the GPU.

© NVIDIA Corporation 2008

NVMEX

- Native MATLAB script cannot parse CUDA code
- New MATLAB script nvmex.m compiles CUDA code (.cu) to create MATLAB function files
- Syntax similar to original mex script:

```
>> nvmex –f nvmexopts.bat filename.cu –IC:\cuda\include
      –LC:\cuda\lib -lcudart
```

Available for Windows and Linux from:

http://developer.nvidia.com/object/matlab_cuda.html

© NVIDIA Corporation 2008

Timing details

1024x1024 mesh, 400 RK4 steps on Windows, 2D isotropic turbulence

	Runtime Opteron 250	Speed up	Runtime Opteron 2210	Speed up
PCI-e Bandwidth: Host to/from device	1135 MB/s 1003 MB/s		1483 MB/s 1223 MB/s	
Standard MATLAB	8098 s		9525s	
Overload FFT2 and IFFT2	4425 s	1.8x	4937s	1.9x
Overload Szeta	735 s	11.x	789s	12.X
Overload Szeta , FFT2 and IFFT2	577 s	14.x	605s	15.7x

© NVIDIA Corporation 2008