M02: High Performance Computing with CUDA

CUDA Toolkit
Massimiliano Fatica
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
CUDA Toolkit

Application Software
Industry Standard C Language

Libraries
- cuFFT
- cuBLAS
- cuDPP

GPU: card, system
- Multicore CPU

CUDA Compiler
- C
- Fortran

CUDA Tools
- Debugger
- Profiler

Multicore CPU

240 cores

4 cores
CUDA for Many-Core and Multi-Core

- CUDA automatically scale across multiple cores
- CUDA expresses parallelism well
- Easy approach to multi-core CPU programming
- CUDA code feeds into standard optimizing CPU compiler
CUDA Many-core + Multi-core support

C CUDA Application

NVCC

Many-core

PTX to Target Compiler

Many-core

NVCC

--multicore

Multi-core

CPU C code

gcc and MSVC

Multi-core
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**)
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
Compiling CUDA for Multi-Core

- Using “—multicore” compile switch with the NVCC compiler generates C code for multi-core CPU
- Performance scales linearly with more cores
- Control numbers of cores with environment variable CUDA_NROF_CORES=n

C/C++ CUDA Application

NVCC --multicore

Multicore CPU C Code

gcc / MSVC

Multicore Optimized Application
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
Closely Coupled CPU-GPU

- Integrated programming model
- High speed data transfer – up to 5.5GB/sec
- Asynchronous data transfer
- Large GPU memory systems
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
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
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
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)
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 -l/usr/local/cuda/include -c fortran.c
g95 -O3 -DCUBLAS code.f90 fortran.o -L/usr/local/cuda/lib -lcublas
```
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,A,m1,B,m1,beta,C,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
DGEMM Performance

![Graph showing DGEMM Performance with different configurations and sizes.](image)
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
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.
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
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:

\[ \text{IFFT}(\text{FFT}(A)) = \text{length}(A) \times A \]

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
Code example: 2D complex to complex transform

```c
#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 1D 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);
```
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.
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.

```fortran
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 Fortran 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
Calling CUDA kernels from FORTRAN

From Fortran call C function that will call CUDA kernel

```fortran
! 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;
    cudaMemcpy ((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
```

cuda_function.o: cuda_function.cu
```
nvcc -c -O3 cuda_function.cu
```
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.
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:
Mex files for CUDA

A typical mex file will perform the following steps:

1. Convert from double to single precision
2. Rearrange the data layout for complex data
3. Allocate memory on the GPU
4. Transfer the data from the host to the GPU
5. Perform computation on GPU (library, custom code)
6. Transfer results from the GPU to the host
7. Rearrange the data layout for complex data
8. Convert from single to double
9. Clean up memory and return results to MATLAB

Some of these steps will go away with new versions of the library (2,7) and new hardware (1,8)
CUDA MEX example

Additional code in MEX file to handle CUDA

/* Parse input, convert to single precision and to interleaved complex format */
    ...
/* Allocate array on the GPU */
cufftComplex *rhs_complex_d;
cudaMalloc( (void **) &rhs_complex_d,sizeof(cufftComplex)*N*M);
/* Copy input array in interleaved format to the GPU */
cudaMemcpy( rhs_complex_d, input_single, sizeof(cufftComplex)*N*M, cudaMemcpyHostToDevice);
/* Create plan for CUDA FFT NB: transposing dimensions*/
cufftPlan2d(&plan, N, M, CUFFT_C2C) ;
/* Execute FFT on GPU */
cufftExecC2C(plan, rhs_complex_d, rhs_complex_d, CUFFT_INVERSE) ;
/* Copy result back to host */
cudaMemcpy( input_single, rhs_complex_d, sizeof(cufftComplex)*N*M, cudaMemcpyDeviceToHost);
/* Clean up memory and plan on the GPU */
cufftDestroy(plan); cudaFree(rhs_complex_d);
/* Convert back to double precision and to split complex format */
## Timing details

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

<table>
<thead>
<tr>
<th></th>
<th>Runtime Opteron 250</th>
<th>Speed up</th>
<th>Runtime Opteron 2210</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCI-e Bandwidth:</td>
<td>1135 MB/s</td>
<td></td>
<td>1483 MB/s</td>
<td></td>
</tr>
<tr>
<td>Host to/from device</td>
<td>1003 MB/s</td>
<td></td>
<td>1223 MB/s</td>
<td></td>
</tr>
<tr>
<td>Standard MATLAB</td>
<td>8098 s</td>
<td></td>
<td>9525s</td>
<td></td>
</tr>
<tr>
<td>Overload FFT2 and IFFT2</td>
<td>4425 s</td>
<td>1.8x</td>
<td>4937s</td>
<td>1.9x</td>
</tr>
<tr>
<td>Overload Szeta</td>
<td>735 s</td>
<td>11.x</td>
<td>789s</td>
<td>12.X</td>
</tr>
<tr>
<td>Overload Szeta, FFT2 and IFFT2</td>
<td>577 s</td>
<td>14.x</td>
<td>605s</td>
<td>15.7x</td>
</tr>
</tbody>
</table>
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.
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 \( u \)
% 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);
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);
Implementation steps

The following steps need to be performed:

1. Allocate memory on host: \( r \) (\( N \times N \)) , \( u \) (\( N \times N \)) , \( k_x \) (\( N \)) and \( k_y \) (\( N \))
2. Allocate memory on device: \( r_d \), \( u_d \), \( k_x_d \), \( k_y_d \)
3. Transfer \( r \), \( k_x \) and \( k_y \) 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
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.
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);
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);
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
/* 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);
/*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);
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;
    }
}
solve_poisson (with shared memory)

```c
__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 (threadIdx.x < 1) kx_s[threadIdx.x] = kx[idx];
    if (threadIdx.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[threadIdx.y]*ky_s[threadIdx.y]);
        if (idx == 0 && idy == 0) scale = 1.f;
        scale = 1.f / scale;
        c[index].x *= scale;
        c[index].y *= scale;
    }
}
```

\[
\hat{\phi} = - \frac{\hat{r}}{\left(k_x^2 + k_y^2\right)}
\]
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/NVIDIA_CUDA_SDK/common/inc
-L/usr/local/NVIDIA_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