CUDA Tools and Libraries

Melbourne HPC GPU Computing Workshop
Mark Harris, NVIDIA    July 2011
CUDA-GDB Debugger

- Extended version of GDB with support for CUDA C
- Supported on Mac and Linux 32bit / 64bit systems
- Seamlessly debug both the host|CPU and device|GPU code
  - Set breakpoints on any source line or symbol name
  - Single step executes only one warp – except on __syncthreads()
  - Access and print all CUDA memory allocations, local, global, constant and shared vars.
Linux GDB Integration with EMACS

```c
__device_func__(float __cuda_acosf(float a))
{
    float t0, t1, t2;
    t0 = __cuda_fabsf(a);
    t2 = 1.0f - t0;
    t2 = __cuda_sqrtf(t2);
    t1 = t0 > 0.5f ? t2 : t0;
    t1 = __internal_asinf_kernel(t1);
    if (!defined(__CUDA__) )
    #if __cuda__isnanf(a) {
        t1 = a + a;
        #endif
        return t1;
    }

    __device_func__(float __cuda_logf(float a))
    {
        #if defined(__MULTI_CORE__)
        return logf(a);
        #elif defined(__USE_FAST_MATH__)
        return __logf(a);
        #else /* __MULTI_CORE__ */
        return __internal_accurate_logf(a);
        #endif /* __MULTI_CORE__ */
    }

    __device_func__(float __cuda_log10f(float a))
    {
        #if defined(__MULTI_CORE__)
        return log10f(a);
        #elif defined(__USE_FAST_MATH__)
        return __log10f(a);
        #else /* __MULTI_CORE__ */
        return __internal_accurate_log10f(a);
        #endif /* __MULTI_CORE__ */
    }
```
Linux GDB Integration with DDD
CUDA-MemCheck

- Detects/tracks memory errors
  - Out of bounds accesses
  - Misaligned accesses (types must be aligned on their size)
- Standalone command-line tool, and integrated into CUDA-GDB
  - Linux, Mac, Windows

```
[jchase@dhcp.1686_Linux_debug]$ cuda-memcheck ./ptrchecktest
Checking...
Done
Checking...
Error: 3 (65538)
Done
Checking...
Error: 0 (1)
Error: 1 (0)
Error: 2 (0)
Done
unspecified launch failure : 125
Invalid read of size 4
---------
             at 0x0000000f0 in kernel2 (/src/gpgpu/cudamemcheck/test/ptrchecktest.cu:27)
             by thread 5 in block 3
---------
Address 0x00101015 is misaligned
---------
Invalid read of size 4
---------
             at 0x0000000f0 in kernel1 (/src/gpgpu/cudamemcheck/test/ptrchecktest.cu:18)
             by thread 3 in block 5
---------
Address 0x00101020 is out of bounds
---------
```
CUDA Driver – Command-line Profiling

1. Set environment variables
   - export CUDA_PROFILE=1
   - export CUDA_PROFILE_CSV=1
   - export CUDA_PROFILE_CONFIG=config.txt
   - export CUDA_PROFILE_LOG=profile.csv

2. Set configuration file
   FILE "config.txt":
   gpustarttimestamp
   instructions

3. Run application
   - matrixMul

4. View profiler output

<table>
<thead>
<tr>
<th>FILE &quot;profile.csv&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td># CUDA_PROFILE_LOG_VERSION 1.5</td>
</tr>
<tr>
<td># CUDA_DEVICE 0 GeForce 8800 GT</td>
</tr>
<tr>
<td># CUDA_PROFILE_CSV 1</td>
</tr>
<tr>
<td># TIMESTAMPFACCTOR fa292bb1ea2c12c</td>
</tr>
<tr>
<td>gpustarttimestamp,method,gputime,cputime,occupancy,instructions</td>
</tr>
<tr>
<td>115f4ea10e3b220,memcpyHtoD,7.328,12.000</td>
</tr>
<tr>
<td>115f4ea10e5dac0,memcpyHtoD,5.664,4.000</td>
</tr>
<tr>
<td>115f4ea10e95ce0,memcpyHtoD,7.328,6.000</td>
</tr>
<tr>
<td>115f4ea10f2ea60,<em>Z10dmatrixmulPfiiS_iis</em>,19.296,40.000,0.333,4352</td>
</tr>
<tr>
<td>115f4ea10f443a0,memcpyDtoH,7.776,36.000</td>
</tr>
</tbody>
</table>
CUDA Visual Profiler

- Performance analysis for CUDA apps
  - Linux, Windows, Mac
- Execute app and collect profiling data
  - Hardware performance counters
  - Profile all kernels and memory xfers
  - Profiling data analysis
Automated Performance Analysis in Visual Profiler

New in CUDA 4.0

Summary analysis & hints

- Session
- Device
- Context
- Kernel

New UI for kernel analysis

- Identify limiting factor
- Analyze instruction throughput
- Analyze memory throughput
- Analyze kernel occupancy
Visual Studio with Parallel Nsight

Integrated development for CPU and GPU
Parallel Nsight for Compute

Parallel Debugger

Debug compute kernels directly on GPU hardware
Examine thousands of threads executing in parallel in Visual Studio
Use conditional breakpoints to correct errors in massively parallel code

System Analysis

Capture CPU and GPU level events on a single correlated timeline
Timeline inspection tools allow for the examination of workload dependencies
Profile CUDA kernels using GPU performance counters
Parallel Nsight for Graphics

**Graphics Debugger**
- Debug HLSL shaders directly on GPU hardware
- Real-time inspection of hardware shaders, DirectX API calls and GPU pipeline state
- See contributing fragments with Pixel History

**Graphics Profiler**
- Examine profile characteristics of a rendered frame
- Filter and sort draw calls by state buckets and performance marker groups
- Use the performance graphs to identify and eliminate bottlenecks

**System Analysis**
- View CPU and GPU events on a single timeline
- Examine workload dependencies
- DirectX3D and OpenGL API Trace
Parallel Nsight Update
For Compute and Parallel Development

- Released May 2011
- CUDA Toolkit 4.0 Support
- Full Visual Studio 2010 Platform Support
- PTX/SASS Assembly Debugging
- Advanced Conditional Breakpoints
- CUDA Attach to Process
- Tesla Compute Cluster (TCC) Analysis
- CUDA Derived Metrics and Experiments
  - Achieved Occupancy
  - Instruction Throughput
- CUDA Concurrent Kernel Trace
- CUDA Runtime API Trace

http://www.nvidia.com/GetParallelNsight
Beyond Version 2.0

Parallel Nsight Update
For Graphics Development

- View HLSL disassembly
- Pre-compiled shader debugging
- GeForce GT 520 GPU support**

** For certain D3D10_* compiler flags. See the user documentation for more details.

http://www.nvidia.com/GetParallelNsight
Parallel Nsight Update
For Compute and Parallel Development

- CUDA Toolkit 4.1 Support
- New CUDA Information Page
- Parallel Locals and Watch Windows
- GPU Break on Assert
- Support for CUDA C/C++ debugging on Optimus
- New CUDA profiling and experiments
  - Divergence and branches
  - Coalesced vs. non-coalesced memory operations
  - Statistics on stalls on memory transactions
- OpenCL 1.1 API Trace Support
  Available Q4 2011
Parallel Nsight Feedback
For Compute and Parallel Development

- CUDA C/C++ source level profiling
- Release debugging of PTX/SASS
- Rule-based analysis of captured data
- Determine kernel bounded-ness
- Visualize when the GPU is idle
- Additional metrics and experiments
  - Warp trace
  - Source correlation
  - Per-line Divergence

And Beyond
How can I learn more about Parallel Nsight?

- Parallel Nsight Documentation
  - Start → All Programs →
  - NVIDIA Parallel Nsight 2.0 → User Guide

- Parallel Nsight Instruction Videos

- CUDA Books and References
  - Programming Massively Parallel Processors
  - CUDA by Example
Tesla Compute Cluster (TCC) Driver

- Registers GPU as a Compute Device
  - vs. as a graphics device

- Enables GPUs for Remote Desktop Protocol (RDP)
  - Remote desktop connections, applications running as services, and Windows Cluster Management

- Delivers Increased Performance
  - Avoids graphics overhead of Windows Display Driver Model (WDDM)

- Now available as configuration mode
  - Use nvidia-smi to control TCC mode for each GPU individually

- Supported on Windows Server 2008 and Windows 7
NVIDIA CUDA Libraries

CUDA Toolkit includes several libraries:

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

Several open source and commercial libraries:

- **MAGMA:** Linear Algebra - **OpenVidia:** Computer Vision
- **CULA Tools:** Linear Algebra - **OpenCurrent:** CFD
- **CUSP:** Sparse Linear Solvers - **CUDPP:** Data-parallel Primitives
- **NAG:** Computational Finance
NVIDIA CUDA Libraries

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

- 3rd Party Libraries

- CUDA C/Fortran

Applications
CUFFT Library

CUFFT is a GPU based Fast Fourier Transform library

\[
F(x) = \sum_{n=0}^{N-1} f(n)e^{-j2\pi(x)^n / N}
\]

\[
f(n) = \frac{1}{N} \sum_{n=0}^{N-1} F(x)e^{j2\pi(x)^n / N}
\]
CUFFT Library Features

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

- **Support of 4 types:**
  - Float, Double, Complex, Double Complex
  - Respective Prefixes: S, D, C, Z

- **Contains 152 routines:** S(37), D(37), C(41), Z(41)

- **Function naming convention:** cublas + BLAS name

- **Example:** cublasSGEMM
  - S: single precision (float)
  - GE: general
  - M: multiplication
  - M: matrix
GEMM Performance

![GEMM Performance on 4K by 4K matrices](chart.png)

- **SGEMM**: 636 GFLOPS
- **CGEMM**: 775 GFLOPS
- **DGEMM**: 301 GFLOPS
- **ZGEMM**: 295 GFLOPS

Performance may vary based on OS version and motherboard configuration. cuBLAS 3.2, Tesla C2050 (Fermi), ECC on MKL 10.2.3, 4-core Core i7 @ 2.66 GHz.
CUSPARSE

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

\[
\begin{bmatrix}
y_1 \\
y_2 \\
y_3 \\
y_4 \\
\end{bmatrix} = \alpha \begin{bmatrix} 1.0 & 2.0 & 5.0 \\ 3.0 & 3.0 & 6.0 \\ 4.0 & 2.0 & 7.0 \\
\end{bmatrix} \begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \\
\end{bmatrix} + \beta \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\
\end{bmatrix}
\]
CUDA standard math library features

High performance and high accuracy implementation:

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

• Continuous enhancements to performance and accuracy

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

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

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

\[ \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-t^2} dt \]
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:**
   curandCreateGenerator()

2. **Set a seed:**
   curandSetPseudoRandomGeneratorSeed()

3. **Generate the data from a distribution:**
   - curandGenerateUniform() / (curandGenerateUniformDouble()): Uniform
   - curandGenerateNormal() / cuRandGenerateNormalDouble(): Gaussian
   - curandGenerateLogNormal / curandGenerateLogNormalDouble(): Log-Normal

4. **Destroy the generator:**
   curandDestroyGenerator()
CURAND Performance

XORWOW Pseudo-RNG

Sobol’ Quasi-RNG (1 dimension)

Performance may vary based on OS version and motherboard configuration

CURAND 3.2, NVIDIA C2050 (Fermi), ECC on
NVIDIA Performance Primitives (NPP)

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

• Data exchange & initialization
  – Set, Convert, CopyConstBorder, Copy, Transpose, SwapChannels

• Arithmetic & Logical Ops
  – Add, Sub, Mul, Div, AbsDiff

• Threshold & Compare Ops
  – Threshold, Compare

• Color Conversion
  – RGB To YCbCr (and 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

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

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

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

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

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

    return 0;
}
```
Algorithms

- **Elementwise operations**
  
  for_each, transform, gather, scatter ...

- **Reductions**
  
  reduce, inner_product, reduce_by_key ...

- **Prefix-Sums**
  
  inclusive_scan, inclusive_scan_by_key ...

- **Sorting**
  
  sort, stable_sort, sort_by_key ...
Thrust Algorithm Performance

Various Algorithms (32M integers)
Speedup compared to std

Sort (32M samples)
Speedup compared to std

*Thrust 4.0, NVIDIA Tesla C2050 (Fermi)  
*Core i7 950 @ 3.07GHz
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!
```
Interoperability (from C/CUDA to Thrust)

- 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);
```
Thrust on Google Code

- http://thrust.googlecode.com
- Quick Start Guide
- Examples
- Documentation
- Mailing List (thrust-users)
Questions?
CUDA Fortran

- PGI / NVIDIA collaboration
- Same CUDA programming model as CUDA-C with Fortran syntax
- Strongly typed – variables with device-type reside in GPU memory
- Use standard allocate, deallocate
- Copy between CPU and GPU with assignment statements:
  
  ```
  GPU_array = CPU_array
  ```
- Kernel loop directives (CUF Kernels) to parallelize loops with device data
program gpu_example
  use cudafor
  real, allocatable :: cpu_array(:,:)
  real, allocatable, device :: gpu_array(:,:)
  type(dim3):: grid_size, block_size
! Use standard allocate for CPU and GPU arrays
allocate(cpu_array(n,m), gpu_array(n,m))
call initialize(cpu_array)
! Move data with simple assignment
gpu_array = cpu_array
! Call CUDA kernel
...
  block_size=dim3(16,16,1)
call gpu_func<<<grid_size, block_size>>>( gpu_array, n,m )
cpu_array = gpu_array
deallocate(cpu_array, gpu_array)
end program
CUDA Fortran example

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

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

end subroutine gpu_func
Computing $\pi$ with CUDA Fortran

$$\pi = 4 \times \left( \frac{\sum \text{red points}}{\sum \text{points}} \right)$$

Simple example:
- Generate random numbers (CURAND)
- Compute sum using of kernel loop directive
- Compute sum using two stages reduction with Cuda Fortran kernels
- Compute sum using single stage reduction with Cuda Fortran kernel
- Accuracy
CUDA Libraries from CUDA Fortran

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

```fortran
interface curandGenerateUniform

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

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

! Copy the data back to CPU to check result later
hostData=deviceData
! Perform the test on GPU using CUF kernel
inside=0
$\text{do } i=1,Nhalf$
\quad if( (deviceData(i)**2+deviceData(i+Nhalf)**2) <= 1._fp_kind ) inside=inside+1
\text{end do}
! Perform the test on CPU
inside_cpu=0
\text{do } i=1,Nhalf
\quad if( (hostData(i)**2+hostData(i+Nhalf)**2) <= 1._fp_kind ) inside_cpu=inside_cpu+1
\text{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)\nSamples=",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 $\pi$

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

<table>
<thead>
<tr>
<th>Compute $\pi$ in single precision (seed 1234)</th>
<th>Compute $\pi$ in single precision (seed 1234567)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samples= 10000  Pi=3.11120009  Error= 0.3039E-01</td>
<td>Samples= 10000  Pi=3.16720009  Error= 0.2561E-01</td>
</tr>
<tr>
<td>Samples= 100000  Pi=3.13632011  Error= 0.5273E-02</td>
<td>Samples= 100000  Pi=3.13919997  Error= 0.2393E-02</td>
</tr>
<tr>
<td>Samples= 1000000  Pi=3.14056396  Error= 0.1029E-02</td>
<td>Samples= 1000000  Pi=3.14109206  Error= 0.5007E-03</td>
</tr>
<tr>
<td>Samples= 10000000  Pi=3.14092445  Error= 0.6683E-03</td>
<td>Samples= 10000000  Pi=3.14106607  Error= 0.5267E-03</td>
</tr>
<tr>
<td>Samples= 100000000  Pi=3.14158082  Error= 0.1192E-04</td>
<td>Samples= 100000000  Pi=3.14139414  Error= 0.1986E-03</td>
</tr>
</tbody>
</table>

Mismatch between CPU/GPU: 78534862 78534859

Where is the error coming from?

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

–Sum of the point inside the circle is done with integers (no issues due to floating point arithmetic)
–Computation of the distance from the origin ($x^2+y^2$), 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>10000</td>
<td>3.16720009</td>
<td>0.2561E-01</td>
</tr>
<tr>
<td>100000</td>
<td>3.13919997</td>
<td>0.2393E-02</td>
</tr>
<tr>
<td>1000000</td>
<td>3.14109206</td>
<td>0.5007E-03</td>
</tr>
<tr>
<td>10000000</td>
<td>3.14106187</td>
<td>0.5267E-03</td>
</tr>
<tr>
<td>100000000</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

- Need to use multiple blocks

- Need to use multiple threads in a block:

- No global synchronization:
  two stages approach, same code for both stages
Parallel Reduction: Sequential Addressing

Step 1
Stride 8
Thread IDs
Values
Step 2
Stride 4
Thread IDs
Values
Step 3
Stride 2
Thread IDs
Values
Step 4
Stride 1
Thread IDs
Values
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
! Compute the final sum with 1 block of 256 threads
! Compute the partial sums with 256 blocks of 512 threads
! Compute the final sum with 1 block of 256 threads
! Compute the partial sums with 256 blocks of 512 threads
! Compute the final sum with 1 block of 256 threads
! Compute the partial sums with 256 blocks of 512 threads
! Compute the final sum with 1 block of 256 threads
! Compute the partial sums with 256 blocks of 512 threads
! Compute the final sum with 1 block of 256 threads
! Compute the partial sums with 256 blocks of 512 threads
! Compute the final sum with 1 block of 256 threads
! Compute the partial sums with 256 blocks of 512 threads
! Compute the final sum with 1 block of 256 threads
Computing \( \pi \) with CUDA Fortran kernels (2/2)

```fortran
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 atomic lock

Instead of storing back the partial sum:

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

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

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

partial(1)=0
call sum<<<<64,256,256*4>>>(deviceData,partial,N)
inside=partial(1)
Calling Thrust from CUDA Fortran

C wrapper for Thrust: csort.cu

```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);
  }
}
```
Calling Thrust from CUDA Fortran

Fortran interface to C wrapper using ISO C Bindings

module thrust

interface thrustsort

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

end interface

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
dallocate(cpuData,gpuData)
!
end program testsort
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)"
!Deallocate arrays
deallocate(cpuData,gpuData)
end program timesort

$ ./timesort
Sorting array of 100000000 single precision
Sorted array in: 194.6642 (ms)
After sorting 7.0585919E-09 1.0318221E-08 1.9398616E-08 3.1738640E-08 4.4078664E-08 0.9999999 0.9999999 1.000000 1.000000 1.000000
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)
program example_sgemm
! Define 3 single precision matrices A, B, C
real, dimension(:,,:),allocatable:: A(:,,:),B(:,,:),C(:, ;)
integer:: n=16
allocate (A(n,n),B(n,n),C(n,n))
! Initialize A, B and C
...
#ifdef CUBLAS
! Call SGEMM in CUBLAS library using THUNKING interface (library takes care of
! memory allocation on device and data movement)
call cublas_SGEMM('n','n', n,n,n,1.,A,n,B,n,1.,C,n)
#else
! Call SGEMM in host BLAS library
 call SGEMM ('n','n',m1,m1,m1,alpha,A,m1,B,m1,beta,C,m1)
#endif
print *,c(n,n)
end program example_sgemm

To use the host BLAS routine:
  g95 -O3 code.f90 -L/usr/local/lib -lblas

To use the CUBLAS routine (fortran_thunking.c is included in the toolkit /usr/local/cuda/src):
  nvcc -O3 -c fortran_thunking.c
  g95 -O3 -DCUBLAS code.f90 fortran_thunking.o -L/usr/local/cuda/lib64 -lcudart -lcublas
SGEMM example (NON-THUNKING)

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

To use the CUBLAS routine (fortran.c is included in the toolkit /usr/local/cuda/src):
    nvcc -O3 -c fortran.c
g95 -O3 code.f90 fortran.o -L/usr/local/cuda/lib64 -lcudart -lcublas