CUDA Libraries and Tools

- **Overview**
- **Scan**, Reduce, Map, sort
- Specialized Libraries: CUBLAS, CUFFT, HONIE, PyCUDA, Thrust, CUDPP, CULA
- Development tools: Nexus, gdb, memcheck, Visual Profiler
CUDA Libraries and Tools

Overarching theme:
  - PROGRAMMER PRODUCTIVITY !!!

Programmer productivity
  - Rapidly develop complex applications
  - Leverage parallel primitives

Encourage generic programming
  - Don’t reinvent the wheel
  - E.g. one reduction to rule them all

High performance
  - With minimal programmer effort
References

- Scan primitives for GPU Computing.
  - Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens

- Presentation on scan primitives by Gary J. Katz based on the particle Parallel Prefix Sum (Scan) with CUDA - Harris, Sengupta and Owens (GPU GEMS Chapter 39)

- Super Computing 2009 CUDA Tools (Cohen)

- Thrust Introduction – Nathan Bell
CUDA Libraries and Tools

GPU Computing Applications

CUDA-Accelerated Libraries

CUDA C
- Over 60,000 developers
- Released 2008
- SDK
- Libraries
- Visual Profiler
- Debugger
- Nexus

OpenCL
- Shipped 1st OpenCL Conformant Driver

Direct Compute
- Microsoft’s GPU Computing API
- Supports all CUDA-Architecture GPUs since G80 (DX10 and future DX11 GPUs)

Fortran
- CUDA Fortran
- PGI Accelerator
- NOAA Fortran bindings (ftoc)
- FLAGON

Python, Java, .NET, ...
- PyCUDA
- JaCUDA
- CUDA.NET
- BSGP
- And more...

NVIDIA GPU
with the CUDA Parallel Computing Architecture
Libraries
CUBLAS
CUFFT
MAGMA
CULA
Thrust
...

Tools
CUDA-gdb
CUDA-Memcheck
CUDA Visual Profiler
Nexus
...

Parallel Primitives

- Scan (Parallel Prefix Sum)
- Map
- Reduce
- Sort
- ....

- (Build algorithms in terms of primitives)
Parallel Primitives: Scan

Prefix-Sum Example

- in: 3 1 7 0 4 1 6 3
- out: 0 3 4 11 11 15 16 22

Trivial Sequential Implementation

```c
void scan(int* in, int* out, int n) {
    out[0] = 0;
    for (int i = 1; i < n; i++)
        out[i] = in[i-1] + out[i-1];
}
```
Parallel Primitives: Scan

Definition:

The scan operation takes a binary associative operator $\oplus$ with identity I, and an array of n elements $[a_0, a_1, \ldots, a_{n-1}]$ and returns the array $[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$

Types – inclusive, exclusive, forward, backward
The all-prefix-sums operation on an array of data is commonly known as *scan*.

The scan just defined is an *exclusive* scan, because each element \( j \) of the result is the sum of all elements up to but *not* including \( j \) in the input array.

In an *inclusive* scan, all elements *including* \( j \) are summed.

- An exclusive scan can be generated from an inclusive scan by shifting the resulting array right by one element and inserting the identity.
- An inclusive scan can be generated from an exclusive scan by shifting the resulting array left and inserting at the end the sum of the last element of the scan and the last element of the input array.
Parallel Primitives

- **Exclusive Scan**
  - in: 3 1 7 0 4 1 6 3
  - out: 0 3 4 11 11 15 16 22

- **Inclusive Scan**
  - in: 3 1 7 0 4 1 6 3
  - out: 3 4 11 11 15 16 22 25
Parallel Primitives

For \( d = 1; d < \log_2 n; d++ \)
for all \( k \) in parallel
    if( \( k \geq 2^d \) )
        \[ x[\text{out}][k] = x[\text{in}][k - 2^{d-1}] + x[\text{in}][k] \]
    else
        \[ x[\text{out}][k] = x[\text{in}][k] \]

Complexity \( O(n\log_2 n) \)

Not very work efficient!
Parallel Primitives

- Goal is a parallel scan that is $O(n)$ instead of $O(n \log_2 n)$

- Solution:
  - Balanced Trees: Build a binary tree on the input data and sweep it to and from the root.

  Binary tree with $n$ leaves has $d=\log_2 n$ levels, each level $d$ has $2^d$ nodes

  One add is performed per node, therefore $O(n)$ add on a single traversal of the tree.
Parallel Primitives

O(n) unsegmented scan

- **Reduce/Up-Sweep**
  
  for(d = 0; d < log₂n-1; d++)
  
  for all k=0; k < n-1; k+=2^d+1 in parallel
  
  x[k+2^{d+1}-1] = x[k+2^d-1] + x[k+2^{d+1}-1]

- **Down-Sweep**

  x[n-1] = 0;
  
  for(d = log₂n – 1; d >=0; d--)
  
  for all k = 0; k < n-1; k += 2^{d+1} in parallel
  
  t = x[k + 2^d - 1]
  
  x[k + 2^d - 1] = x[k + 2^{d+1} -1]
  
  x[k + 2^{d+1} - 1] = t + x[k + 2^{d+1} - 1]
Parallel Primitives

Tree analogy

The tree we build is not an actual data structure, but a concept we use to determine what each thread does at each step of the traversal.
Parallel Primitives
Parallel Primitives

- Up-Sweep (Reduce)
  - traverse the tree from leaves to root computing partial sums at internal nodes of the tree.
  - This is also known as a parallel reduction, because after this phase, the root node (the last node in the array) holds the sum of all nodes in the array.

```
1: for \( d' = 0 \) to \( \log_2 n - 1 \) do
2:     for all \( k = 0 \) to \( n - 1 \) by \( 2^{d'+1} \) in parallel do
3:         \( x[k + 2^{d'+1} - 1] = x[k + 2^d - 1] + x[k + 2^d + 1 - 1] \)
```
Parallel Primitives

**Down-Sweep**
- Traverse back down the tree from the root, using the partial sums from the reduce phase to build the scan in place on the array.
- We start by inserting zero at the root of the tree, and on each step, each node at the current level passes its own value to its left child, and the sum of its value and the former value of its left child to its right child.

```
1: \( x[n-1] \leftarrow 0 \)
2: for \( d = \log_2 n - 1 \) down to 0 do
3:   for all \( k = 0 \) to \( n - 1 \) by \( 2^d + 1 \) in parallel do
4:     \( t \leftarrow x[k + 2^d - 1] \)
5:     \( x[k + 2^d - 1] \leftarrow x[k + 2^d + 1 - 1] \)
6:     if \( f_i[k + 2^d] \) is set then
7:       \( x[k + 2^d + 1 - 1] \leftarrow 0 \)
8:     else if \( f[k + 2^d - 1] \) is set then
9:       \( x[k + 2^d + 1 - 1] \leftarrow t \)
10: else
11:     \( x[k + 2^d + 1 - 1] \leftarrow t + x[k + 2^d + 1 - 1] \)
12: Unset flag \( f[k + 2^d - 1] \)
```
Addressing Without Padding

Offset = 1: Address (ai) stride is 2, resulting in 2-way bank conflicts

```
int ai = offset*(2*thid+1)-1;
ext bi = offset*(2*thid+2)-1;
temp[bi] += temp[ai]
```

Bank
---
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

ai thid
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Offset = 2: Address (ai) stride is 4, resulting in 4-way bank conflicts

```
int ai = offset*(2*thid+1)-1;
ext bi = offset*(2*thid+2)-1;
ai += ai / NUM_BANKS;
bi += bi / NUM_BANKS;
temp[bi] += temp[ai]
```

Bank
---
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

ai thid
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Addressing With Padding

Offset = 1: Padding addresses every 16 elements removes bank conflicts

```
int ai = offset*(2*thid+1)-1;
ext bi = offset*(2*thid+2)-1;
ai += ai / NUM_BANKS;
bi += bi / NUM_BANKS;
temp[bi] += temp[ai]
```

Bank
---
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

ai thid
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Offset = 2: Padding addresses every 16 elements removes bank conflicts

```
int ai = offset*(2*thid+1)-1;
ext bi = offset*(2*thid+2)-1;
ai += ai / NUM_BANKS;
bi += bi / NUM_BANKS;
temp[bi] += temp[ai]
```

Bank
---
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

ai thid
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Padding Increment: 0 1 2 3
Parallel Primitives

Features of segmented scan

- 3 times slower than unsegmented scan
- Useful for building broad variety of applications which are not possible with unsegmented scan.

- A convenient way to execute a scan independently over many sets of values
- Inputs: A data vector and a flag vector
- A flag marks the first element of a segment

\[
\begin{align*}
a &= [5, 1, 3, 4, 3, 9, 2, 6] \\
f &= [1, 0, 1, 0, 0, 0, 1, 0] \\
\text{segmented } +\text{-scan} &= [5, 6, 3, 7, 10, 19, 2, 8] \\
\text{segmented max-scan} &= [5, 5, 3, 4, 4, 9, 2, 6]
\end{align*}
\]
Primitives built on scan

- **Enumerate**
  - enumerate([t f f t f t t]) = [0 1 1 1 2 2 3]
  - Exclusive scan of input vector

- **Distribute (copy)**
  - distribute ([a b c] [d e]) = [a a a] [d d]
  - Inclusive scan of input vector

- **Split and split-and-segment**
  
  Split divides the input vector into two pieces, with all the elements marked false on the left side of the output vector and all the elements marked true on the right.
Applications

- Quicksort
- Sparse Matrix-Vector Multiply
- Tridiagonal Matrix Solvers and Fluid Simulation
- Radix Sort
- Stream Compaction
- Summed-Area Tables
Quicksort

[5 3 7 4 6]  # initial input
[5 5 5 5 5]  # distribute pivot across segment
[f f t f t]  # input > pivot?
[5 3 4] [7 6]  # split-and-segment
[5 5 5] [7 7]  # distribute pivot across segment
[t f f] [t f]  # input >= pivot?
[3 4 5] [6 7]  # split-and-segment, done!
Radix Sort Using Scan

<table>
<thead>
<tr>
<th>100</th>
<th>111</th>
<th>010</th>
<th>110</th>
<th>011</th>
<th>101</th>
<th>001</th>
<th>000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Input Array

- b = least significant bit
- e = Insert a 1 for all false sort keys
- f = Scan the 1s
- Total Falses = e[n-1] + f[n-1]
- t = index - f + Total Falses
- d = b ? t : f

Scatter input using d as scatter address
CUDA Specialized Libraries
CUDA Specialized Libraries: CUBLAS

- Cuda Based Linear Algebra Subroutines
- Saxpy, conjugate gradient, linear solvers.
- 3D reconstruction of planetary nebulae example.
CUDA accelerated BLAS (Basic Linear Algebra Subprograms)

- Create matrix and vector objects in GPU memory space
- Fill objects with data
- Call sequence of CUBLAS functions
- Retrieve data from GPU (optionally)

```c
while( i++ < max_iter && deltanew > stop_tol )
{
    cublasSgemv('n', N, N, 1.0, d_A, N, d_d, 1, 0, d_y, 1);
    float alpha = deltanew / cublasSdot(N,d_d,1,d_y,1);
    cublasSaxpy(N, alpha,d_d,1,d_x,1);

    // every 50 iterations, restart residual
    if (i % 50 == 0) {
        cublasSgemv('n', N, N, 1.0, d_A, N, d_x, 1, 0, d_y, 1);
        cublasScopy(N, d_b, 1, d_r, 1);
        cublasSaxpy(N, -1.0, d_y, 1, d_r, 1);
    } else
        cublasSaxpy(N,-alpha,d_y,1,d_r,1);
...
CUBLAS Features

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, DROT, DSCAL, DSWAP, ISAMAX, IDAMIN
- Level 2: DGEMV, DGER, DSYR, DTRSV
- Level 3: ZGEMM, DGEMM, DTRSM, DTRMM, DSYMM, DSYRK, DSYR2K
CUBLAS: Performance – CPU vs GPU

**Single Precision BLAS: SGEMM**

- Gflops vs Matrix Size
  - Tesla C1060

**Double Precision BLAS: DGEMM**

- Gflops vs Matrix Size
  - Tesla C1060

---

CUBLAS: CUDA 2.3, Tesla C1060
MKL 10.0.3: Intel Core2 Extreme, 3.00GHz
CUBLAS

- GPU Variant 100 times faster than CPU version
- Matrix size is limited by graphics card memory and texture size.
- Although taking advantage of sparse matrices will help reduce memory consumption, sparse matrix storage is not implemented by CUBLAS.
CUDA Specialized Libraries: CUFFT

- Cuda Based Fast Fourier Transform Library.
- The FFT is a divide-and-conquer algorithm for efficiently computing discrete Fourier transforms of complex or real-valued data sets,
- One of the most important and widely used numerical algorithms, with applications that include computational physics and general signal processing.
CUFFT

- CUFFT is the CUDA FFT library
- Computes parallel FFT on an NVIDIA GPU
- Uses ‘Plans’ like FFTW
  - Plan contains information about optimal configuration for a given transform.
  - Plans can be persisted to prevent recalculation.
  - Good fit for CUFFT because different kinds of FFTs require different thread/block/grid configurations.
CUFFT

- No. of elements < 8192 slower than fftw
- > 8192, 5x speedup over threaded fftw and 10x over serial fftw.

- 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.
#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);

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

cudaFree(idata);
cudaFree(odata);
CUFFT: Performance – CPU vs GPU

**Single Precision FFT**

- cuFFT 2.3
- MKL 4 Threads
- FFTW 1 Thread

**Double Precision FFT**

- cuFFT 2.3
- MKL 4 Threads

CUFFT 2.3: NVIDIA Tesla C1060 GPU
MKL 10.1r1: Quad-Core Intel Core i7 (Nehalem) 3.2GHz
CUDA Specialized Libraries: MAGMA

- Matrix Algebra on GPU and Multicore Architectures
- The MAGMA project aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with current "Multicore+GPU" systems.
MAGMA: Matrix Algebra on GPU and Multicore Architectures

**MAGMA and LAPACK**
- MAGMA - based on LAPACK, extended for heterogeneous systems
- MAGMA - similar to LAPACK in functionality, data storage, interface

**Features**
- **Goal**: easy porting from LAPACK to take advantage of the new GPU + multicore architectures
- **Leverage**: experience developing open source Linear Algebra software (LAPACK, ScaLAPACK, BLAS, ATLAS)
- **Incorporate**: newest numerical developments (e.g. communication avoiding algorithms) and experiences on homogeneous multicores (e.g. PLASMA)

**MAGMA Developers**
- University of Tennessee, Knoxville
- University of California, Berkeley
- University of Colorado, Denver
- Number of contributors from the LA community

© NVIDIA Corporation 2009

Portions of this slide courtesy Stan Tomov
MAGMA Release

MAGMA version 0.1 (08/04/09)
- One-sided factorizations [for linear solvers] in single and double precision arithmetic
- Hardware target: 1 core + 1 GPU (CUDA enabled)

MAGMA version 0.2 (11/14/09)
- One-sided factorizations in complex arithmetic
- Two-sided factorizations for eigenvalue solvers
- Linear solvers, including least squares and mixed precision iterative solvers
- MAGMA BLAS (gemm optimized for rectangular matrices, triangular solvers, gemv, etc)
- Hardware target:
  - 1 core + 1 GPU (all)
  - multicore + multi-GPU (one-sided factorizations)

Portions of this slide courtesy Stan Tomov
MAGMA Version 0.1 Performance

QR factorization in single precision arithmetic, CPU interface

Performance of MAGMA vs MKL

MAGMA QR time breakdown

GPU : NVIDIA GeForce GTX 280 (240 cores @ 1.30GHz)
CPU : Intel Xeon dual socket quad-core (8 cores @2.33 GHz)

GPU BLAS : CUBLAS 2.2, sgemm peak: 375 GFlop/s
CPU BLAS : MKL 10.0 , sgemm peak: 128 GFlop/s

For more performance data, see http://icl.cs.utk.edu/magma

© NVIDIA Corporation 2009

 Portions of this slide courtesy Stan Tomov
MAGMA Version 0.2 Performance

Linear Solvers
[e.g. \( A x = b \) using LU Factorization]

Hessenberg factorization
[e.g. double precision, CPU interface]

For more performance data, see http://icl.cs.utk.edu/magma

Portions of this slide courtesy Stan Tomov
MAGMA Multi-GPU Performance

Cholesky factorization in single precision arithmetic
Performance and scalability on 4 GPUs

MAGNUM-tiles approach
- Communication-avoiding/tile type algorithms on large (magnum) tiles
- A magnum tile/task is defined for hybrid 1 CPU + 1 GPU computing
- PLASMA scheduling on the magnum tiles/tasks

GPU: NVIDIA Tesla C1070 (4 GPUs @1.44GHz)
CPU: AMD Opteron dual socket dual-core (4 cores @1.8 GHz)

For more performance data, see http://icl.cs.utk.edu/magma

© NVIDIA Corporation 2009
CUDA Specialized Libraries: CULA

- CULA is EM Photonics' GPU-accelerated numerical linear algebra library that contains a growing list of LAPACK functions.
- LAPACK stands for Linear Algebra PACKage. It is an industry standard computational library that has been in development for over 15 years and provides a large number of routines for factorization, decomposition, system solvers, and eigenvalue problems.
3rd Party Implementation of LAPACK interface from EM Photonics (www.culatools.com)

**CULA basic**
- Six popular single/complex-single LAPACK functions
- Free!

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getrf</td>
<td>LU decomposition</td>
</tr>
<tr>
<td>gesv</td>
<td>System solve</td>
</tr>
<tr>
<td>geqrf</td>
<td>QR factorization</td>
</tr>
<tr>
<td>gesvd</td>
<td>Singular value</td>
</tr>
<tr>
<td></td>
<td>decomposition</td>
</tr>
<tr>
<td>gels</td>
<td>Least squares</td>
</tr>
<tr>
<td>gglse</td>
<td>Constrained least</td>
</tr>
<tr>
<td></td>
<td>squares</td>
</tr>
</tbody>
</table>

**CULA premium**
- Available for purchase
- Adds 18 more routines (and growing)
- Adds Double (D) / Double Complex (Z)

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>potrf</td>
<td>Cholesky factorization</td>
</tr>
<tr>
<td>gebrd</td>
<td>Bidiagonalization</td>
</tr>
<tr>
<td>getri</td>
<td>Matrix inversion</td>
</tr>
<tr>
<td>getrs</td>
<td>LU Backsolve</td>
</tr>
<tr>
<td>trtrs</td>
<td>Triangular solve</td>
</tr>
<tr>
<td>gelqf</td>
<td>LQ factorization</td>
</tr>
<tr>
<td>posv</td>
<td>Positive-definite system</td>
</tr>
<tr>
<td></td>
<td>solve</td>
</tr>
</tbody>
</table>

© NVIDIA Corporation 2009

Courtesy EM Photonics
CULA Performance

CULA 1.0 vs Intel MKL 10.2

- gesvd: 7.2x
- geqrf: 5.3x
- gesv: 3.5x
- getrf: 4.2x
- gglse: 4.3x
- gels: 3.9x

Speed Up vs MKL

CULA 1.0 vs Netlib Reference LAPACK

- gesvd: 87.7x
- geqrf: 56.8x
- gesv: 77.1x
- getrf: 92.1x
- gglse: 77.5x
- gels: 63.6x

Speed Up vs Netlib

Tesla C1060 vs Intel Core i7, matrix size ~10,000x10,000

Courtesy EM Photonics
CUDA Specialized Libraries: PyCUDA

- PyCUDA lets you access Nvidia’s CUDA parallel computation API from Python
PyCUDA

- 3rd party open source, written by Andreas Klöckner
- Exposes all of CUDA via Python bindings
- Compiles CUDA on the fly
- presents CUDA as an interpreted language
- Integration with numpy
- Handles memory management, resource allocation
- CUDA programs are Python strings
  - Metaprogramming – modify source code on-the-fly
  - Like a really complex pre-processor
- http://mathema.tician.de/software/pycuda
PyCUDA - Differences

- Object cleanup tied to lifetime of objects. This idiom, often called RAII in C++, makes it much easier to write correct, leak- and crash-free code. PyCUDA knows about dependencies, too, so (for example) it won’t detach from a context before all memory allocated in it is also freed.
- Convenience. Abstractions like pycuda.driver.SourceModule and pycuda.gpucarray.GPUArray make CUDA programming even more convenient than with Nvidia’s C-based runtime.
- Completeness. PyCUDA puts the full power of CUDA’s driver API at your disposal, if you wish.
- Automatic Error Checking. All CUDA errors are automatically translated into Python exceptions.
- Speed. PyCUDA’s base layer is written in C++, so all the niceties above are virtually free.
```python
import pycuda.driver as cuda
import pycuda.autoinit
import numpy

a = numpy.random.randn(4,4).astype(numpy.oat32)
a_gpu = cuda.mem_alloc(a.size, a.dtype.itemsize)
cuda.memcpy_htod(a_gpu, a)

mod = cuda.SourceModule(''
__global__ void doublify(float *a)
{
    int idx = threadIdx.x + threadIdx.y*4;
a[ idx ] *= 2.0f;
}
''
)
func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```
Metaprogramming

In GPU scripting, GPU code does not need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
CUDA Specialized Libraries: CUDPP

- CUDPP: CUDA Data Parallel Primitives Library
  - CUDPP is a library of data-parallel algorithm primitives such as parallel prefix-sum ("scan"), parallel sort and parallel reduction.
CUDPP – Design Goals

- **Performance**: aims to provide best-of-class performance for simple primitives.
- **Modularity**: primitives easily included in other applications.
  - CUDPP is provided as a library that can link against other applications.
  - CUDPP calls run on the GPU on GPU data. Thus they can be used as standalone calls on the GPU (on GPU data initialized by the calling application) and, more importantly, as GPU components in larger CPU/GPU applications.
- CUDPP is implemented as 4 layers:
  - The **Public Interface** is the external library interface, which is the intended entry point for most applications. The public interface calls into the Application-Level API.
  - The **Application-Level API** comprises functions callable from CPU code. These functions execute code jointly on the CPU (host) and the GPU by calling into the Kernel-Level API below them.
  - The **Kernel-Level API** comprises functions that run entirely on the GPU across an entire grid of thread blocks. These functions may call into the CTA-Level API below them.
  - The **CTA-Level API** comprises functions that run entirely on the GPU within a single Cooperative Thread Array (CTA, aka thread block). These are low-level functions that implement core data-parallel algorithms, typically by processing data within shared (CUDA __shared__) memory.
- Programmers may use any of the lower three CUDPP layers in their own programs by building the source directly into their application. However, the typical usage of CUDPP is to link to the library and invoke functions in the CUDPP Public Interface, as in the simpleCUDPP, satGL, and cudpp_testrig application examples included in the CUDPP distribution.
CUDPP

CUDPP_DLL CUDPPResult cudppSparseMatrixVectorMultiply(CUDPPHandle sparseMatrixHandle, void * d_y, const void * d_x)

Perform matrix-vector multiply $y = A \times x$ for arbitrary sparse matrix $A$ and vector $x$. 
CUDPP - Example

CUDPPScanConfig config;
    config.direction = CUDPP_SCAN_FORWARD; config.exclusivity = CUDPP_SCAN_EXCLUSIVE; config.op = CUDPP_ADD;
    config.datatype = CUDPP_FLOAT; config.maxNumElements = numElements; config.maxNumRows = 1;
    config.rowPitch = 0;

    cudppInitializeScan(&config);

    cudppScan(d_odata, d_idata, numElements, &config);
CUDA Specialized Libraries: HONEI

- A collection of libraries for numerical computations targeting multiple processor architectures
HONEI, an open-source collection of libraries offering a hardware oriented approach to numerical calculations.

HONEI abstracts the hardware, and applications written on top of HONEI can be executed on a wide range of computer architectures such as CPUs, GPUs and the Cell processor.

- The most important frontend library is libhoneila, HONEI's linear algebra library. It provides templated container classes for different matrix and vector types.
- The numerics and math library libhoneimath contains high performance kernels for iterative linear system solvers as well as other useful components like interpolation and approximation.
CUDA Specialized Libraries: Thrust

- Thrust is a CUDA library of parallel algorithms with an interface resembling the C++ Standard Template Library (STL). Thrust provides a flexible high-level interface for GPU programming that greatly enhances developer productivity. Develop high-performance applications rapidly with Thrust!
- “Standard Template Library for CUDA”
- Heavy use of C++ templates for efficiency
Facts and Figures

- **Thrust v1.0**
  - Open source (Apache license)
  - 1,100+ downloads

- **Development**
  - 460+ unit tests
  - 25+ compiler bugs reported
  - 35k lines of code
    - Including whitespace & comments

- **Uses CUDA Runtime API**
  - Essential for template generation
What is Thrust?

- C++ template library for CUDA
  - Mimics Standard Template Library (STL)
- Containers
  - thrust::host_vector<T>
  - thrust::device_vector<T>
- Algorithms
  - thrust::sort()
  - thrust::reduce()
  - thrust::inclusive_scan()
  - thrust::segmented_inclusive_scan()
  - Etc.
Containers

- Make common operations concise and readable
  - Hides `cudaMalloc` & `cudaMemcpy`

```cpp
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host vector to device
thrust::device_vector<int> d_vec = h_vec;

// manipulate device values from the host
d_vec[0] = 13;
d_vec[1] = 27;

std::cout << "sum: " << d_vec[0] + d_vec[1] << std::endl;
```
Containers

- Compatible with STL containers
  - Eases integration
  - vector, list, map, ...

```cpp
// list container on host
std::list<int> h_list;
h_list.push_back(13);
h_list.push_back(27);

// copy list to device vector
thrust::device_vector<int> d_vec(h_list.size());
thrust::copy(h_list.begin(), h_list.end(), d_vec.begin());

// alternative method
thrust::device_vector<int> d_vec(h_list.begin(), h_list.end());
```
Iterators

- Track memory space (host/device)
  - Guides algorithm dispatch

```cpp
// initialize random values on host
thrust::host_vector<int> h_vec(1000);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

// copy values to device
thrust::device_vector<int> d_vec = h_vec;

// compute sum on host
int h_sum = thrust::reduce(h_vec.begin(), h_vec.end());

// compute sum on device
int d_sum = thrust::reduce(d_vec.begin(), d_vec.end());
```
Algorithms

- Thrust provides ~50 algorithms
  - Reduction
  - Prefix Sums
  - Sorting

- Generic definitions
  - General Types
    - builtin types (int, float, ...)
    - User-defined structures
  - General Operators
    - reduce with plus(a,b)
    - scan with maximum(a,b)
Algorithms

General types and operators

// declare storage
device_vector<int> i_vec = ...
device_vector<float> f_vec = ...

// sum of integers (equivalent calls)
reduce(i_vec.begin(), i_vec.end());
reduce(i_vec.begin(), i_vec.end(), 0, plus<int>());

// sum of floats (equivalent calls)
reduce(f_vec.begin(), f_vec.end());
reduce(f_vec.begin(), f_vec.end(), 0.0f, plus<float>());

// maximum of integers
reduce(i_vec.begin(), i_vec.end(), 0, maximum<int>());
Halftime Summary

- **Containers**
  - Manage host & device memory
    - Automatic allocation and deallocation
    - Simplify data transfers

- **Iterators**
  - Behave like pointers
  - Associated with memory spaces

- **Algorithms**
  - Generic
    - Work for any type or operator
  - Statically dispatched based on iterator type
    - Memory space is known at compile-time
Fancy Iterators

- Behave like “normal” iterators
  - Algorithms don't know the difference

- Examples
  - constant_iterator
  - counting_iterator
  - transform_iterator
  - zip_iterator
Fancy Iterators

- constant_iterator
  - An infinite array filled with a constant value

```c++
// create iterators
constant_iterator<int> first(10);
constant_iterator<int> last = first + 3;

first[0]    // returns 10
first[1]    // returns 10
first[100]  // returns 10

// sum of [first, last)
reduce(first, last);  // returns 30 (i.e. 3 * 10)
```
Fancy Iterators

- **counting_iterator**
  - An infinite array with sequential values

```cpp
// create iterators
counting_iterator<int> first(10);
counting_iterator<int> last = first + 3;

first[0]   // returns 10
first[1]   // returns 11
first[100] // returns 110

// sum of [first, last)
reduce(first, last);   // returns 33 (i.e. 10 + 11 + 12)
```
Fancy Iterators

- transform_iterate
  - Yields a transformed sequence
  - Facilitates kernel fusion

F( x )

F( X ) F( Y ) F( Z )
Fancy Iterators

- **transform_iterator**
  - Conserves memory capacity and bandwidth

```cpp
// initialize vector
device_vector<int> vec(3);

// create iterator (type omitted)
first = make_transform_iterator(vec.begin(), negate<int>());
last = make_transform_iterator(vec.end(), negate<int>());

first[0]  // returns -10
first[1]  // returns -20

// sum of [first, last)
reduce(first, last);  // returns -60 (i.e. -10 + -20 + -30)
```
Fancy Iterators

- **zip_iterator**
  - Looks like an array of structs (AoS)
  - Stored in structure of arrays (SoA)
**Fancy Iterators**

- **zip_iterator**

  ```cpp
  // initialize vectors
  device_vector<int> A(3);
  device_vector<char> B(3);

  // create iterator (type omitted)
  first = make_zip_iterator(make_tuple(A.begin(), B.begin()));
  last  = make_zip_iterator(make_tuple(A.end(),   B.end()));

  first[0] // returns tuple(10, 'x')
  first[1] // returns tuple(20, 'y')
  first[2] // returns tuple(30, 'z')

  // maximum of [first, last)
  maximum< tuple<int,char> > binary_op;
  reduce(first, last, first[0], binary_op); // returns tuple(30, 'z')
  ```
Features & Optimizations

- **gather & scatter**
  - Works between host and device

- **fill & reduce**
  - Avoids G8x coalescing rules for char, short, etc.

- **sort**
  - Dispatches radix_sort for all primitive types
    - Uses optimal number of radix_sort iterations
  - Dispatches merge_sort for all other types
Examples

- **SNRM2**
  - Computes norm of a vector
  - Level 1 BLAS function

- **2D Bucket Sort**
  - Sorting points into cells of a 2D grid
  - Compute bounds for each bucket
Example: SNRM2

// define transformation f(x) -> x^2
struct square
{
    __host__ __device__
        float operator()(float x) { return x * x; }
};

// setup arguments
square      unary_op;
plus<float> binary_op;
float init = 0;

// initialize vector
device_vector<float>  A(3);

// compute norm
float norm = sqrt( transform_reduce(A.begin(), A.end(),
                                        unary_op, init, binary_op) );
Example: 2D Bucket Sort

Procedure:

[Step 1] create random points
[Step 2] compute bucket index for each point
[Step 3] sort points by bucket index
[Step 4] compute bounds for each bucket
Example: 2D Bucket Sort

[Step 1] create random points

```cpp
// number of points
const size_t N = 100000;

// return a random float2 in [0,1)^2
float2 make_random_float2(void)
{
    return make_float2(rand() / (RAND_MAX + 1.0f),
                        rand() / (RAND_MAX + 1.0f));
}

// allocate some random points in the unit square on the host
host_vector<float2> h_points(N);
generate(h_points.begin(), h_points.end(), make_random_float2);

// transfer to device
device_vector<float2> points = h_points;
```
Example: 2D Bucket Sort

[Step 2] compute bucket index for each point

```c
struct point_to_bucket_index
{
    unsigned int w, h;

    __host__ __device__
    point_to_bucket_index(unsigned int width, unsigned int height)
    :w(width), h(height){}

    __host__ __device__
    unsigned int operator()(float2 p) const
    {
        // coordinates of the grid cell containing point p
        unsigned int x = p.x * w;
        unsigned int y = p.y * h;

        // return the bucket's linear index
        return y * w + x;
    }
};
```
Example: 2D Bucket Sort

[Step 2] compute bucket index for each point

// resolution of the 2D grid
unsigned int w = 200;
unsigned int h = 100;

// allocate storage for each point's bucket index
device_vector<unsigned int> bucket_indices(N);

// transform the points to their bucket indices
transform(points.begin(),
    points.end(),
    bucket_indices.begin(),
    point_to_bucket_index(w,h));
Example: 2D Bucket Sort

[Step 3] sort points by bucket index

```cpp
// sort the points by their bucket index
sort_by_key(bucket_indices.begin(),
            bucket_indices.end(),
            points.begin());
```

![Diagram showing points sorted by bucket index]
Example: 2D Bucket Sort

[Step 4] compute bounds for each bucket

```cpp
// bucket_begin[i] indexes the first element of bucket i
// bucket_end[i] indexes one past the last element of bucket i
device_vector<unsigned int> bucket_begin(w*h);
device_vector<unsigned int> bucket_end(w*h);

// used to produce integers in the range [0, w*h)
counting_iterator<unsigned int> search_begin(0);

// find the beginning of each bucket's list of points
lower_bound(bucket_indices.begin(), bucket_indices.end(),
            search_begin, search_begin + w*h, bucket_begin.begin());

// find the end of each bucket's list of points
upper_bound(bucket_indices.begin(), bucket_indices.end(),
            search_begin, search_begin + w*h, bucket_end.begin());
```
CUDA Development Tools
CUDA Development Tools: CUDA-gdb

- Simple Debugger integrated into gdb
CUDA-gdb

- Integrated into gdb
- Supports CUDA C
- Seamless CPU+GPU development experience
- Enabled on all CUDA supported 32/64bit Linux distros
- Set breakpoint and single step any source line
- Access and print all CUDA memory allocs, local, global, constant and shared vars.
} else {
    acos_noftz_main<<ACOS_CTA_CNT,ACOS_THREAD_CNT>>>(funcParams);
}
}
}

} else {
    if (opts.ieee == 3) {
        acos_ieee3_ftz_main<<ACOS_CTA_CNT,ACOS_THREAD_CNT>>>(funcParams);
    } else if (opts.ieee == 2) {
        acos_ieee2_ftz_main<<ACOS_CTA_CNT,ACOS_THREAD_CNT>>>(funcParams);
    } else if (opts.ieee == 1) {
        acos_ieee1_ftz_main<<ACOS_CTA_CNT,ACOS_THREAD_CNT>>>(funcParams);
    } else {
        acos_main<<ACOS_CTA_CNT,ACOS_THREAD_CNT>>>(funcParams);
    }
}

#endif /* FERMI */

acos_main<<ACOS_CTA_CNT,ACOS_THREAD_CNT>>>(funcParams);

}

} else {
    if (cudaStat != cudaSuccess) {
        fprintf (stderr, "!!!!! program launch failed\n");
        CLEANUP();
        return EXIT_FAILURE;
    }
    t0 = __cuda_fabsf(a);
    t2 = 1.0f - t0;
    t2 = 0.5f * t2;
    t2 = __cuda_sqrtf(t2);
    t1 = t0 > 0.5f ? t2 : t0;
    t1 = __internal_asinf_kernel(t1);
    t1 = t0 > 0.5f ? 2.0f * t1 : CUDARTPIO2F - t1;
    if (__cuda__signbitf(a)) {
        t1 = CUDARTPI_F - t1;
    }
    if (!defined(__CUDABE__))
        if (__cuda__isnanf(a)) {
            t1 = a + a;
        }
    #endif
    return t1;
}
```c
/* ------------------ target code ------------------*/

global__ void acos_main (struct acosParams parms)
{
  int i;
  int totalThreads = blockDim.x * blockDim.x;
  int ctaStart = blockDim.x * blockIdx.x;
  for (i = ctaStart + threadIdx.x; i < parms.n; i += totalThreads) {
    parms.res[i] = acosl(parms.arg[i]);
  }
}
```
CUDA Development Tools : MemCheck

- Track memory accesses
CUDA-MemCheck

- Coming with CUDA 3.0 Release
- Track out of bounds and misaligned accesses
- Supports CUDA C
- Integrated into the CUDA-GDB debugger
- Available as standalone tool on all OS platforms.
[jchase@dhcp-172-16-175-68 i686_Linux_debug]$ cuda-memcheck ./ptrchecktest
======== CUDA-MEMCHECK
Checking...
Done
Checking...
Error: 3 (65538)
Done
Checking...
Error: 0 (1)
Error: 1 (0)
Error: 2 (0)
Error: 3 (0)
Error: 4 (0)
Error: 5 (0)
Error: 6 (0)
Error: 7 (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 0x00101028 is out of bounds
========
======== Invalid write of size 8
======== at 0x000000170 in kernel3 (/src/gpgpu/cudamemcheck/test/ptrchecktest.cu:38)
======== by thread 1 in block 8
======== Address 0x00102004 is misaligned
========
======== Invalid write of size 4
======== at 0x000000a0 in kernel4 (/src/gpgpu/cudamemcheck/test/ptrchecktest.cu:44)
======== by thread 63 in block 22
======== Address 0x00000000 is out of bounds
========
ERROR SUMMARY: 4 errors
[jchase@dhcp-172-16-175-68 i686_Linux_debug]$
CUDA Development Tools : Visual Profiler

- Profile your CUDA code
CUDA Visual Profiler
Events are tracked with hardware counters on signals in the chip:

- **timestamp**
- **gld_incoherent**
- **gld_coherent**
- **gst_incoherent**
- **gst_coherent**
  - Global memory loads/stores are coalesced (coherent) or non-coalesced (incoherent) (Compute 1.0/1.1)
- **local_load**
- **local_store**
  - Local loads/stores
- **branch**
- **divergent_branch**
  - Total branches and divergent branches taken by threads
- **instructions** – instruction count
- **warpSerialize** – thread warps that serialize on address conflicts to shared or constant memory
- **cta_launched** – executed thread blocks
CUDA Development Tools : Nexus

- IDE for GPU Computing on Windows: Code Named Nexus
NVIDIA IDE: code name “Nexus”

The first development environment for massively parallel applications.

- Hardware GPU Source Debugging
- Platform-wide Analysis
- Complete Visual Studio-integration

Register for the Beta!

Releasing in Q1 2010
Nexus