The thrust library
Atomic operations in CUDA

October 16, 2015
Quote of the Day

“The whole problem with the world is that fools and fanatics are always so certain of themselves, but wiser people so full of doubts.”

-- Bertrand Russell (1872-1970)
[U.K., philosopher]
Before We Get Started

- Issues covered last time:
  - Memory access patterns (must be coalesced and aligned)

- Today’s topics
  - Wrap up, GPU parallel computing w/ the **thrust** library [started by Hammad]
  - Atomic operations

- Assignment:
  - HW05 – due on Oct. 21 at 11:59 PM
Global Memory Access Issues
### General Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Code</th>
</tr>
</thead>
</table>
| Unary Transformation | `for (int i = 0; i < N; i++)
X[i] = f(A[i]);`                                                    |
| Binary Transformation | `for (int i = 0; i < N; i++)
X[i] = f(A[i],B[i]);`                                                  |
| Ternary Transformation | `for (int i = 0; i < N; i++)
X[i] = f(A[i],B[i],C[i]);`                                              |
| General Transformation | `for (int i = 0; i < N; i++)
X[i] = f(A[i],B[i],C[i],...);`                                      |

- Like C++ STL, **thrust** provides built-in support for unary and binary transformations
- Transformations involving 3 or more input ranges must use a different approach
General Transformations Preamble:
The Zipping Operation

Multiple Distinct Sequences

Unique Sequence of Tuples

NVIDIA [N. Bell]
Example: General Transformations

```cpp
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

struct linear_combo {
    __host__ __device__ float operator()(thrust::tuple<float,float,float> t) {
        float x, y, z;
        thrust::tie(x,y,z) = t;
        return 2.0f * x + 3.0f * y + 4.0f * z;
    }
};

int main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3);
    thrust::device_vector<float> U(3);


    thrust::transform
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
         U.begin(),
         linear_combo());

    for (size_t i = 0; i < Z.size(); i++)
        std::cout << "U[" << i << "] = " << U[i] << "\n";
    return 0;
}
```

Functor Definition

These are the important parts: three different entities are zipped together in one big one.
Example: thrust::transform_reduce

```cpp
#include <thrust/transform_reduce.h>
#include <thrust/device_vector.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>

struct linear_combo {
    __host__ __device__
    float operator()(thrust::tuple<float, float, float> t) {
        float x, y, z;
        thrust::tie(x, y, z) = t;
        return 2.0f * x + 3.0f * y + 4.0f * z;
    }
};

int main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3), U(3);


    thrust::plus<float> binary_op;
    float init = 0.f;

    float myResult = thrust::transform_reduce
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
         linear_combo(),
         init,
         binary_op);

    std::cout << myResult << std::endl;
    return 0;
}
```
thrust, Efficiency Issues
[fusing transformations]
Performance Considerations
[short detour: 1/3]

- Picture below shows key parameters
  - Peak flop rate
  - Max bandwidth

NVIDIA [N. Bell]→

![Diagram of Tesla C2050 with key parameters](image-url)
Arithmetic Intensity
[short detour: 2/3]

- SAXPY
- FFT
- SGEMM

Memory bound

Compute bound

FLOP/Byte

NVIDIA [N. Bell]→
## Arithmetic Intensity

### [short detour: 3/3]

<table>
<thead>
<tr>
<th>Kernel</th>
<th>FLOP/Byte*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Addition</td>
<td>1 : 12</td>
</tr>
<tr>
<td>SAXPY</td>
<td>2 : 12</td>
</tr>
<tr>
<td>Sum</td>
<td>1 : 4</td>
</tr>
<tr>
<td>Max Index</td>
<td>1 : 12</td>
</tr>
</tbody>
</table>

* excludes indexing overhead

<table>
<thead>
<tr>
<th>Hardware**</th>
<th>FLOP/Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce GTX 280</td>
<td>~7.0 : 1</td>
</tr>
<tr>
<td>GeForce GTX 480</td>
<td>~7.6 : 1</td>
</tr>
<tr>
<td>Tesla C870</td>
<td>~6.7 : 1</td>
</tr>
<tr>
<td>Tesla C1060</td>
<td>~9.1 : 1</td>
</tr>
<tr>
<td>Tesla C2050</td>
<td>~7.1 : 1</td>
</tr>
</tbody>
</table>

** lists the number of flop per byte of data to reach peak Flop/s rate

“Byte” refers to a Global Memory byte
Fusing Transformations

for (int i = 0; i < N; i++)
    U[i] = F(X[i], Y[i], Z[i]);

for (int i = 0; i < N; i++)
    V[i] = G(X[i], Y[i], Z[i]);

for (int i = 0; i < N; i++)
{
    U[i] = F(X[i], Y[i], Z[i]);
    V[i] = G(X[i], Y[i], Z[i]);
}

Loop Fusion

- One way to look at things...
  - Zipping: reorganizes data for thrust processing
  - Fusing: reorganizes computation for efficient thrust processing

- **NOTE**: Fusing is a general purpose idea (not specific to thrust)
typedef thrust::tuple<float, float> Tuple2;
typedef thrust::tuple<float, float, float> Tuple3;

struct linear_combo {
    __host__ __device__
    Tuple2 operator()(Tuple3 t) {
        float x, y, z; thrust::tie(x, y, z) = t;
        float u = 2.0f * x + 3.0f * y + 4.0f * z;
        float v = 1.0f * x + 2.0f * y + 3.0f * z;
        return Tuple2(u, v);
    }
};

int main(void) {
    thrust::device_vector<float> X(3), Y(3), Z(3);
    thrust::device_vector<float> U(3), V(3);

    thrust::transform
        (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin(), Z.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(X.end(), Y.end(), Z.end())),
         thrust::make_zip_iterator(thrust::make_tuple(U.begin(), V.begin())),
         linear_combo());

    return 0;
}
Fusing Transformations

Original Implementation

Optimized Implementation

- Since the operation is completely memory bound the expected speedup is ~1.6x (=32/20)
Fusing Transformations

for (int i = 0; i < N; i++)
    Y[i] = F(X[i]);

for (int i = 0; i < N; i++)
    sum += Y[i];

for (int i = 0; i < N; i++)
    sum += F(X[i]);

Loop Fusion
Fusing Transformations

```cpp
#include <thrust/device_vector.h>
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>
#include <iostream>

using namespace thrust::placeholders;

int main(void) {
    thrust::device_vector<float> X(3);

    float result = thrust::transform_reduce
        (X.begin(), X.end(),
         _1 * _1,
         0.0f,
         thrust::plus<float>())
        ;

    std::cout << "sum of squares is " << result << "\n";
    return 0;
}
```

NVIDIA [N. Bell] →
Fusing Transformations

Original Implementation

Optimized Implementation

- Try to answer this: how many times will we be able to run faster if we fuse?
Maximum Index (Optimized)

Original Implementation

- GPU
- 8 Bytes
- 4 Bytes
- DRAM

Optimized Implementation

- GPU
- 4 Bytes
- DRAM

- Try to answer this: how many times will we be able to run faster if we fuse?
Good Speedups Compared to Multi-threaded CPU Execution

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
thrust Wrap-Up

• Provides boost in productivity at the price of small performance penalty
  • No need to be aware of execution configuration, shared memory, etc.

• Key concepts
  • Functor
  • Zipping data
  • Fusing operations

• Why not always use thrust?
  • There is no “solve multibody dynamics” support in thrust.
  • Focuses only on primitives, provides building blocks
    • The idea: use these building blocks to develop domain-specific solutions
thrust on Google Code

- Quick Start Guide
- Examples
- News
- Documentation
- Mailing List (thrust-users)
thrust in “GPU Computing Gems”

This chapter demonstrates how to leverage the Thrust parallel template library to implement high-performance applications with minimal programming effort. Based on the C++ Standard Template Library (STL), Thrust brings a familiar high-level interface to the realm of GPU Computing while remaining fully interoperable with the rest of the CUDA software ecosystem. Applications written with Thrust are concise, readable, and efficient.

26.1 MOTIVATION

With the introduction of CUDA C/C++, developers can harness the massive parallelism of the GPU through a standard programming language. CUDA allows developers to make fine-grained decisions about how computations are decomposed into parallel threads and executed on the device. The level of control offered by CUDA C/C++ (henceforth CUDA C) is an important feature: it facilitates the development of high-performance algorithms for a variety of computationally demanding tasks which (1) merit significant optimization and (2) profit from low-level control of the mapping onto hardware.

For this class of computational tasks CUDA C is an excellent solution. Thrust (1) solves a complementary set of problems, namely those that are (1) implemented efficiently without a detailed mapping of work onto the target architecture or those that (2) do not merit or simply will not receive significant optimization effort by the user. With Thrust, developers describe their computation using a collection of high-level algorithms and completely delegate the decision of how to implement the computation to the library. This abstract interface allows programmers to describe what to compute without placing any additional restrictions on how to carry out the computation. By constraining the programmer’s intent at a high level, Thrust has the discretion to make informed decisions about problem decomposition, data types, and data layout. The end result is that the library can generate code tailored to the GPU architecture that is both concise, readable, and efficient.

PDF available at http://goo.gl/adj9S
Example, **thrust**: Processing Rainfall Data

Rain situation, end of first day, for a set of five observation stations. Results, summarized over a period of time, reported in the table below.

<table>
<thead>
<tr>
<th>day</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>5</th>
<th>6</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>site</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>measurement</td>
<td>9</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>10</td>
<td>...</td>
</tr>
</tbody>
</table>

Remarks:
1) Time series sorted by day
2) Measurements of zero are excluded from the time series
### Example: Processing Rainfall Data

<table>
<thead>
<tr>
<th>day</th>
<th>[0 0 1 2 5 5 6 6 7 8 ... ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>site</td>
<td>[2 3 0 1 1 2 0 1 2 1 ... ]</td>
</tr>
<tr>
<td>measuremen</td>
<td>[9 5 6 3 3 8 2 6 5 10 ... ]</td>
</tr>
</tbody>
</table>

- Given the data above, here’re some questions you might ask:
  - Total rainfall at a given site
  - Total rainfall between given days
  - Total rainfall on each day
  - Number of days with any rainfall
Total Rainfall at a Given Site

```cpp
struct one_site_measurement
{
    int siteOfInterest;

    one_site_measurement(int site) : siteOfInterest(site) {}

    __host__ __device__
    int operator()(thrust::tuple<int,int> t)
    {
        if (thrust::get<0>(t) == siteOfInterest)
            return thrust::get<1>(t);
        else
            return 0;
    }
};

template <typename Vector>
int compute_total_rainfall_at_one_site(int siteID, const Vector& site, const Vector& measurement)
{
    return thrust::transform_reduce
        (thrust::make_zip_iterator(thrust::make_tuple(site.begin(), measurement.begin())),
         thrust::make_zip_iterator(thrust::make_tuple(site.end(), measurement.end())),
         one_site_measurement(siteID),
         0,
         thrust::plus<int>())
;
}
```
Total Rainfall Between Given Days

```
template <typename Vector>
int compute_total_rainfall_between_days(int first_day, int last_day,
    const Vector& day, const Vector& measurement)
{
    int first = thrust::lower_bound(day.begin(), day.end(), first_day) - day.begin();
    int last  = thrust::upper_bound(day.begin(), day.end(), last_day) - day.begin();

    return thrust::reduce(measurement.begin() + first, measurement.begin() + last);
}
```

For this to fly, you’ll need to include several header files (not all for the code snippet above)

```cpp
#include <thrust/device_vector.h>
#include <thrust/binary_search.h>
#include <thrust/transform.h>
#include <thrust/iterator/zip_iterator.h>
#include <iostream>
```
template<typename Vector>
int compute_number_of_days_with_rainfall(const Vector& day)
{
    return thrust::inner_product(day.begin(), day.end() - 1, 
    day.begin() + 1, 
    0, 
    thrust::plus<int>(), 
    thrust::not_equal_to<int>()) + 1;
}
template <typename Vector>
void compute_total_rainfall_per_day(const Vector& day, const Vector& measurement, 
Vector& day_output, Vector& measurement_output)
{
    size_t N = compute_number_of_days_with_rainfall(day); //see previous slide

day_output.resize(N);
measurement_output.resize(N);

thrust::reduce_by_key(day.begin(), day.end(),
    measurement.begin(),
    day_output.begin(),
    measurement_output.begin());
}
Coda Parallelization via Directives...
3 Ways to Accelerate on GPU

Application

Libraries

Directives

Programming Languages

Easiest Approach  Maximum Performance

Direction of increased performance (and effort)
OpenACC

- Seeks to become:
  - A standard for directives-based Parallel Programming
  - Provide portability across hardware platforms and compiler vendors

- Promoted by NVIDIA, Cray, CAPS, PGI
OpenACC Specification

- Hardware agnostic and platform independent (CPU only, different GPUs)
- OpenACC is an open standard for directives based computing
- Announced at SC11 [November 2011]
- Caps, Cray, and PGI shipping OpenACC Compilers as of Q1 2012
  - AMD and NVIDIA also promoting use of OpenACC
- A version 2.0 draft is out for comments, idea is to expand OpenMP with features that enables one to farm out work to the GPUs
The OpenACC Idea

- Host code computes an approximation for π:

```cpp
#include <iostream>
#include <math.h>
using namespace std;

int main( int argc, char *argv[] )
{
    const double PI25DT = 3.141592653589793;
    const int n=1000000;
    double h = 1.0 / (double) n;
    double sum = 0.0;

    for( int i=0; i<=n; i++ )
    {
        double x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    double mypi = h * sum;

    cout << "Approx. value: " << mypi << endl;
    cout << "Error: " << fabs(mypi-PI25DT) << endl;
    return 0;
}
```
The OpenACC Idea

- Code computes an approximation for \( \pi \) [might use multi-core or GPU]

```cpp
#include <iostream>
#include <math.h>
using namespace std;

int main(int argc, char *argv[])
{
    const double PI25DT = 3.141592653589793238462643;

    const int n = 1000000;
    double h = 1.0 / (double) n;
    double sum = 0.0;
    // # pragma acc region for
    for (int i = 0; i <= n; i++) {
        double x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    double mypi = h * sum;

    cout << "Approx. value: " << mypi << endl;
    cout << "Error: " << fabs(mypi - PI25DT) << endl;
    return 0;
}
```

Add one line of code (a directive): provides a hint to the compiler about opportunity for parallelism.
Atomic Operations
Choreographing Memory Operations

- Accesses to shared locations (global memory & shared memory) need to be correctly coordinated (orchestrated) to avoid race conditions.

- In many common shared memory multithreaded programming models, one uses coordination mechanisms such as locks to choreograph accesses to shared data.

- CUDA has a scalable coordination mechanism called “atomic memory operation”.

Next slides: several examples that show “race conditions”
Race Condition

- A contrived (artificial) example...

```c
// update.cu
__global__ void update_race(int* x, int* y)
{
    int i = threadIdx.x;
    if (i < 2)
        *x = *y;
    else
        *x += 2*i;
}

// main.cpp
update_race<<<1,4>>>(d_x, d_y);
cudaMemcpy(y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```
Relevant Issue: Thread Divergence in “if-then-else”

- Handling of an `if-then-else` construct in CUDA
  - First a subset of threads of the warp execute the “then” branch
  - Next, the rest of the threads in the warp execute the “else” branch
    - Note: done if there is an “else” branch
Question: what happens if in the previous example we change the “if(i<2)” to “if(i>=2) ...” and swap the then and the else parts?

- Is the outcome any different?
- Any difference in behavior compared to sequential computing?
```c
#include <cuda.h>
#include "stdio.h"

__global__ void testKernel(int *x, int *y) {
    int i = threadIdx.x;
    if (i == 0) *x = 1;
    if (i == 1) *y = *x;
}

int main() {
    int* dArr;
    int hArr[2] = {23, -5};
    cudaMemcpy(&dArr, 2 * sizeof(int));
    cudaMemcpy(dArr, hArr, 2 * sizeof(int), cudaMemcpyHostToDevice);
    testKernel <<<1, 2 >>>(dArr, dArr + 1);
    cudaMemcpy(hArr, dArr, 2 * sizeof(int), cudaMemcpyDeviceToHost);
    printf("x = %d\n", hArr[0]);
    printf("y = %d\n", hArr[1]);
    return 0;
}
Example: Inter-Block Issue

- Would this fly?

```c++
// update.cu
__global__ void update(int* x, int* y)
{
    int i = threadIdx.x;
    if (i == 0) *x = blockIdx.x;
    if (i == 1) *y = *x;
}

// main.cpp
update<<<2,5>>>(d_x, d_y);
cudaMemcpy(y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```
Atomic memory operations (atomic functions) are used to solve coordination problems in parallel computer systems.

General concept: provide a mechanism for a thread to update a memory location such that the update appears to happen atomically (without interruption) with respect to other threads.

This ensures that all atomic updates issued concurrently are performed (often in some unspecified order) and that all threads can observe all updates.
Atomic Functions

Atomic functions perform read-modify-write operations on data residing in global and shared memory.

```c
//example of int atomicAdd(int* addr, int val)
__global__ void update(unsigned int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j = atomicAdd(x, i);  // j is now old value of x;
}
```

```c
// snippet of code in main.cpp
int x = 0;
cudaMemcpy(&d_x, &x, cudaMemcpyHostToDevice);
update<<<1,128>>>(x_d);
cudaMemcpy(&x, &d_x, cudaMemcpyDeviceToHost);
```

Atomic functions guarantee that only one thread may access a memory location while the operation completes.

Order in which threads get to write is not specified though…
Atomic Functions

Atomic functions perform read-modify-write operations on data that can reside in global or shared memory.

Synopsis of atomic function \texttt{atomicOP}(a,b) is typically

\begin{verbatim}
t1 = *a;        // read
t2 = (*a) OP (*b); // modify
*a = t2;        // write
return t1;
\end{verbatim}

The hardware ensures that all statements are executed atomically without interruption by any other atomic functions.

The atomic function returns the initial value, *not* the final value, stored at the memory location.
The name atomic is used because the update is performed atomically: it cannot be interrupted by other atomic updates.

The order in which concurrent atomic updates are performed is not defined, and may appear arbitrary.

While order is not clear, none of the atomic updates will be lost.

Several different kinds of atomic operations:
- Add (add), Sub (subtract), Inc (increment), Dec (decrement)
- And (bit-wise and), Or (bit-wise or), Xor (bit-wise exclusive or)
- Exch (Exchange)
- Min (Minimum), Max (Maximum)
- Compare-and-Swap
A Histogram Example

// Compute histogram of colors in an image
//
// picturePixels – pointer to picture pixels, each w/ its own color
// bucket – pointer to histogram bucket of size equal to # of colors
//
__global__ void histogram(int n, int* picturePixels, int* bucket)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n)
    {
        int c = picturePixels[i];
        atomicAdd(&bucket[c], 1);
    }
}
Performance Notes

- Atomics are slower than normal accesses (loads, stores)

- Performance can degrade when many threads attempt to perform atomic operations on a small number of locations

- Possible to have all threads on the machine stalled, waiting to perform atomic operations on a single memory location

- Atomics: convenient to use, come at a typically high efficiency loss…
Important note about Atomics

- Atomic updates are not guaranteed to appear atomic to concurrent accesses using loads and stores

```c
__global__ void broken(int n, int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i == 0)
    {
        *x = *x + 1;
    }
    else
    {
        int j = atomicAdd(x, 1); // j = *x; *x += i;
    }
}

// main.cpp
broken<<<1,128>>>(128, d_x); // d_x = d_x + {1, 127, 128}
```
Summary of Atomics

- When to use: Cannot use normal load/store because of possible race conditions
- Use for infrequent, sparse, and/or unpredictable global communication
- Use shared memory and/or customized data structures & algorithms to avoid synchronization whenever reasonable
- Recent compute capabilities (3 and above) implement very fast atomics
As far as CUDA is concerned, there is a qualitative difference between a \texttt{__syncthreads()} function and an atomic operation.

- \texttt{__syncthreads()} has the connotation of barrier; i.e., of synchronization.
  - \texttt{__syncthreads()} establishes a point in the execution of the kernel that every thread in the **block** needs to reach before any block thread can move beyond that point.

- The “atomic operation” concept instead tied to the idea of coordination in relation to operations that involve memory transactions.
  - Threads in a grid of blocks coordinate their execution so that a certain memory operation invoked in a kernel is conducted in an atomic fashion.