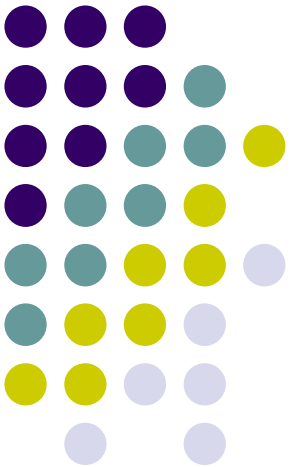
ME964
High Performance Computing for Engineering Applications

Wrap-up, Streams in CUDA

GPU computing using `thrust`

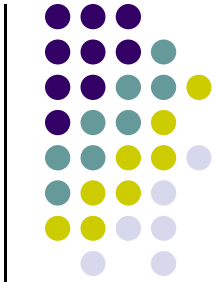
March 22, 2012



"How is education supposed to make me feel smarter? Besides, every time I learn something new, it pushes some old stuff out of my brain. Remember when I took that home winemaking course, and I forgot how to drive?"
Homer Simpson

© Dan Negrut, 2012
ME964 UW-Madison

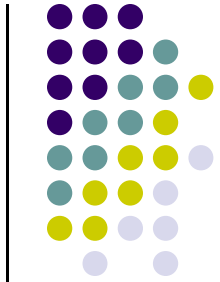
Before We Get Started…



- Last time
  - Discuss two parallel implementations of the prefix scan operation [related to assignment]
  - Asynchronous concurrent execution with CUDA
  - CUDA “streams”
    - Handling multiple streams in CUDA as a means to enable task parallelism

- Today
  - Wrap up CUDA streams and applications
  - GPU programming with **`thrust`**

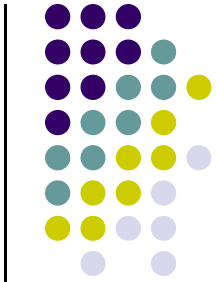
- Other issues
  - Due date for current assignment now Sunday, March 25 at 11:59 PM
  - Assignment 9 posted today or tomorrow, due on Sunday, April 1 at 23:59 pm
    - Related to **`thrust`**
  - Preliminary Midterm Project due on Th, March 29 at 11:59 PM
    - Two page description of [the actual problem you solve] + [software&algorithm design solution]
    - Need to know what’s input & what’s output, how the output is obtained from the input (the algorithm), and how the algorithm is mapped onto the underlying hardware (software design)



CUDA Streams

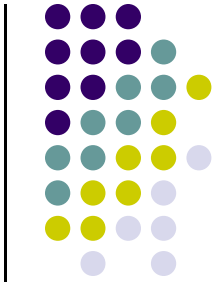
[cntd., from previous lecture]

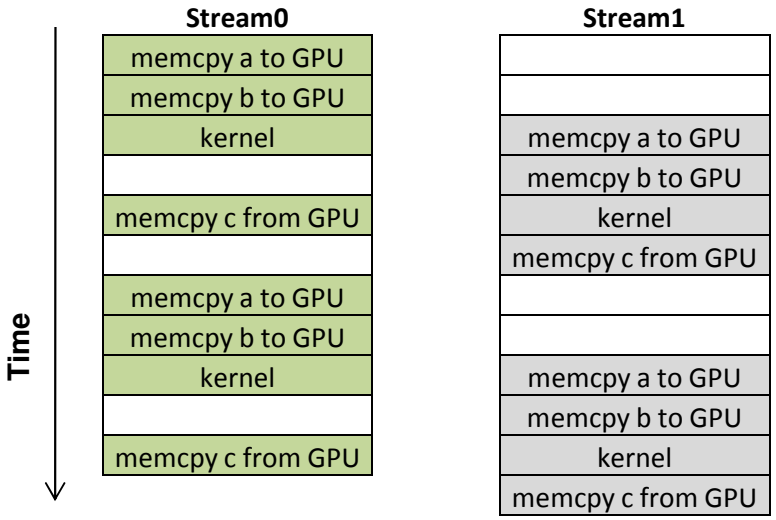
Example 2: Using Multiple Streams
[Version 1]



- Implement the same example but use two streams to this end
- Why would you want to use multiple streams?
  - For devices that are capable of overlapping execution with host↔device data movement, you might hide this data movement and improve overall performance
- Two ideas underlie the process
  - The idea of "chunkification" of the computation
    - Large computation is broken into pieces that are queued up for execution on the device (we already saw this in Example 1, which uses one stream)
  - The idea of overlapping execution with host↔device data movement
  - NOTE: I didn't want to call this tiling, although it's similar to that. However, "tiling" is something that happens exclusively on the device (from global to shared memory). Here, the "chunkification" happens on the host

Overlapping Execution and Data Transfer: A Desirable Scenario

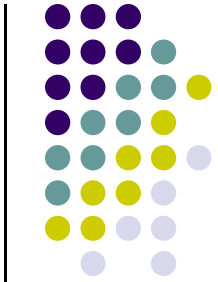




Timeline of intended application execution using two independent streams

- Observations:
  - "memcpy" actually represents an asynchronous `cudaMemcpyAsync()` memory copy call
  - White (empty) boxes represent time when one stream is waiting to execute an operation that it cannot overlap with the other stream's operation
  - The goal: keep both GPU engine types (execution and mem copy) busy
    - Note: recent hardware allows two copies to take place simultaneously: one from host to device, at the same time one goes on from device to host (you have two copy engines)

The "main()" Function, Two Streams



```
01| int main( void ) {
02|     cudaDeviceProp  prop;
03|     int whichDevice;
04|     HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
05|     HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
06|     if (!prop.deviceOverlap) {
07|         printf( "Device will not handle overlaps, so no speed up from streams\n" );
08|         return 0;
09|     }
```

Stage 1

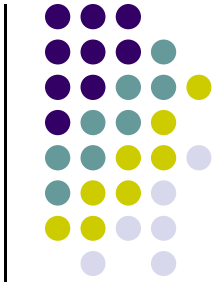
```
11|     cudaEvent_t     start, stop;
12|     float           elapsedTime;
13|
14|     cudaStream_t    stream0, stream1;
15|     int *host_a, *host_b, *host_c;
16|     int *dev_a0, *dev_b0, *dev_c0;
17|     int *dev_a1, *dev_b1, *dev_c1;
18|
19|     // start the timers
20|     HANDLE_ERROR( cudaEventCreate( &start ) );
21|     HANDLE_ERROR( cudaEventCreate( &stop ) );
22|
23|     // initialize the streams
24|     HANDLE_ERROR( cudaStreamCreate( &stream0 ) );
25|     HANDLE_ERROR( cudaStreamCreate( &stream1 ) );
26|
```

Stage 2

Stage 3

```
27|     // allocate the memory on the GPU
28|     HANDLE_ERROR( cudaMalloc( (void**)&dev_a0, N * sizeof(int) ) );
29|     HANDLE_ERROR( cudaMalloc( (void**)&dev_b0, N * sizeof(int) ) );
30|     HANDLE_ERROR( cudaMalloc( (void**)&dev_c0, N * sizeof(int) ) );
31|     HANDLE_ERROR( cudaMalloc( (void**)&dev_a1, N * sizeof(int) ) );
32|     HANDLE_ERROR( cudaMalloc( (void**)&dev_b1, N * sizeof(int) ) );
33|     HANDLE_ERROR( cudaMalloc( (void**)&dev_c1, N * sizeof(int) ) );
34|
35|     // allocate host locked memory, used to stream
36|     HANDLE_ERROR( cudaHostAlloc( (void**)&host_a, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
37|     HANDLE_ERROR( cudaHostAlloc( (void**)&host_b, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
38|     HANDLE_ERROR( cudaHostAlloc( (void**)&host_c, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
```

The "main()" Function, Two Streams
[Cntd.]



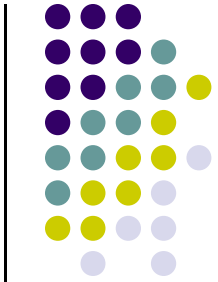
Still Stage 3

```
39|    for (int i=0; i<FULL_DATA_SIZE; i++) {
40|        host_a[i] = rand();
41|        host_b[i] = rand();
42|    }
43|
44|    HANDLE_ERROR( cudaEventRecord( start, 0 ) );
```

Stage 4

```
45|    // now loop over full data, in bite-sized chunks
46|    for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
47|        // copy the locked memory to the device, async
48|        HANDLE_ERROR( cudaMemcpyAsync( dev_a0, host_a+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );
49|        HANDLE_ERROR( cudaMemcpyAsync( dev_b0, host_b+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );
50|
51|        kernel<<<(N+255),256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );
52|
53|        // copy the data from device to locked memory
54|        HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c0, N * sizeof(int), cudaMemcpyDeviceToHost, stream0 ) );
55|
56|
57|        // copy the locked memory to the device, async
58|        HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );
59|        HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );
60|
61|        kernel<<<N/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );
62|
63|        // copy the data from device to locked memory
64|        HANDLE_ERROR( cudaMemcpyAsync( host_c+i+N, dev_c1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1 ) );
65| }
66|
```

The "main()" Function, Two Streams
[Cntd.]

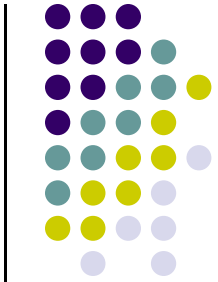


```
67|    HANDLE_ERROR( cudaStreamSynchronize( stream0 ) );
68|    HANDLE_ERROR( cudaStreamSynchronize( stream1 ) );
69|
70|    HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
71|
72|    HANDLE_ERROR( cudaEventSynchronize( stop ) );
73|    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime, start, stop ) );
74|    printf( "Time taken:  %3.1f ms\n", elapsedTime );
75|
76|    // cleanup the streams and memory
77|    HANDLE_ERROR( cudaFreeHost( host_a ) );
78|    HANDLE_ERROR( cudaFreeHost( host_b ) );
79|    HANDLE_ERROR( cudaFreeHost( host_c ) );
80|    HANDLE_ERROR( cudaFree( dev_a0 ) );
81|    HANDLE_ERROR( cudaFree( dev_b0 ) );
82|    HANDLE_ERROR( cudaFree( dev_c0 ) );
83|    HANDLE_ERROR( cudaFree( dev_a1 ) );
84|    HANDLE_ERROR( cudaFree( dev_b1 ) );
85|    HANDLE_ERROR( cudaFree( dev_c1 ) );
86|    HANDLE_ERROR( cudaStreamDestroy( stream0 ) );
87|    HANDLE_ERROR( cudaStreamDestroy( stream1 ) );
88|
89|    return 0;
90|}
```

Stage 5

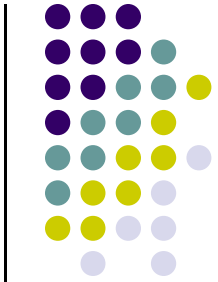
NOTE: the kernel doesn't actually change…

Example 2 [Version 1], Summary



- Stage 1 ensures that your device supports your attempt to overlap kernel execution with host↔device data transfer
- Stage 2 sets up the events needed to time the execution of the program
- Stage 3 allocates page-locked memory on the host side so that we can fall back on asynchronous memory copy operations between host and device and initializes data
- Stage 4 enques the set of GPU operations that need to be undertaken (the “chunkification”)
- Stage 5 takes care of timing reporting and clean up

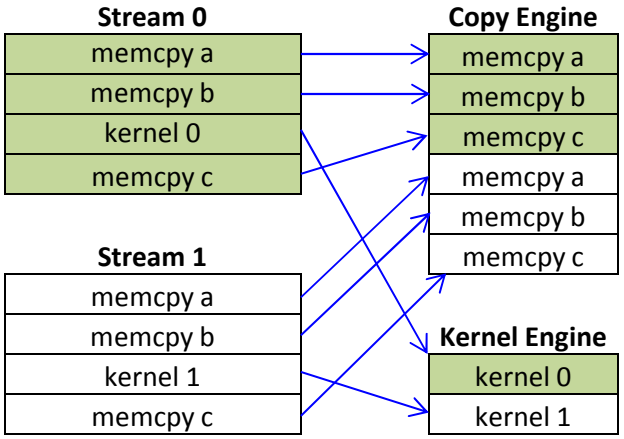
Comments, Using Two Streams
[Version 1]



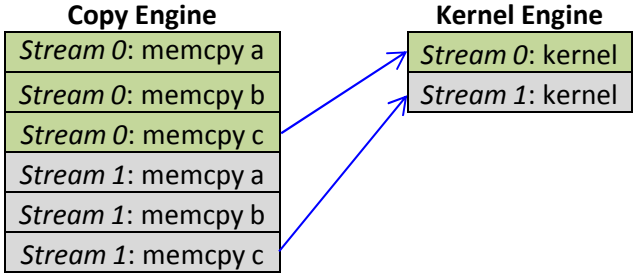
- Timing results provided by “CUDA by Example: An Introduction to General-Purpose GPU Programming,”
  - Sanders and Kandrot reported results on NVIDIA GTX285
- Using one stream (in Example 1): 62 ms
- Using two streams (this example, version 1): 61 ms
- Lackluster performance goes back to the way the two GPU engines (kernel execution and copy) are scheduled

The Two Stream Example, Version 1

Looking Under the Hood



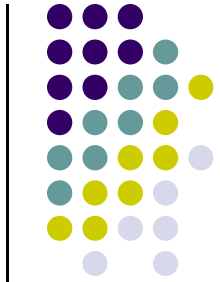
Mapping of CUDA streams onto GPU engines

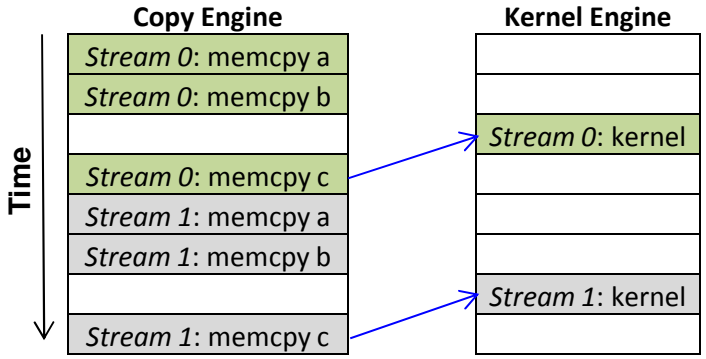


Arrows depicting the dependency of cudaMemcpyAsync () calls on kernel executions in the 2 Streams example

- At the left:
  - An illustration of how the work queued up in the streams ends up being assigned by the CUDA driver to the two GPU engines (copy and execution)
  - Important remark: FIFO is also observed in relation to scheduling the engines (not only the streams)
- At the right
  - Image shows dependency that is implicitly set up in the two streams given the way the streams were defined in the code
  - The queue in the Copy Engine, combined with the dependencies defined determines the scheduling of the Copy and Kernel Engines (see next slide)

The Two Stream Example
Looking Under the Hood



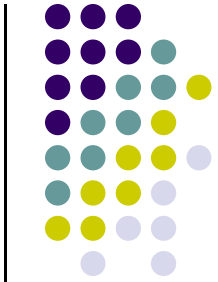


Execution timeline of the 2 Stream example (blue line shows dependency; empty boxes represent idle segments)

- Note that due to the *specific* way in which the streams were defined (depth first), basically there is no overlap of copy & execution…
  - Explains the no net-gain in efficiency compared to the one stream example
- Remedy: go breadth first, instead of depth first
  - In the current version, execution on the two engines was inadvertently blocked by the way the streams have been set up and the existing scheduling and lack of dependency checks available in the current version of CUDA

The Two Stream Example

[Version 2: A More Effective Implementation: Breadth First]

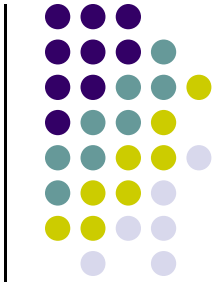


- Old way (the depth first approach):
  - Assign the copy of a, copy of b, kernel execution, and copy of c to stream0. Subsequently, do the same for stream1

- New way (the breadth first approach):
  - Add the copy of a to stream0, and then add the copy of a to stream1
  - Next, add the copy of b to stream0, and then add the copy of b to stream1
  - Next, enqueue the kernel invocation in stream0, and then enqueue one in stream1.
  - Finally, enqueue the copy of c back to the host in stream0 followed by the copy of c in stream1.

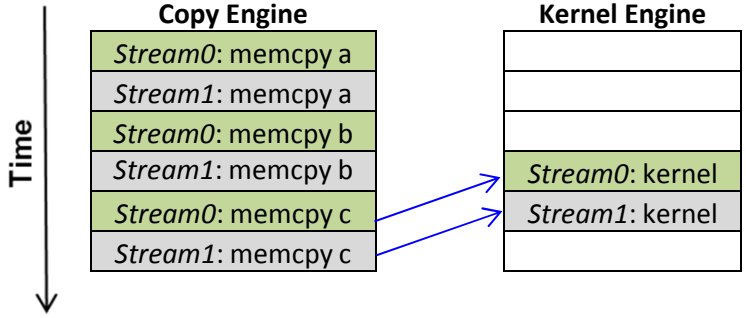
The Two Stream Example

A 20% More Effective Implementation (48 vs. 61 ms)



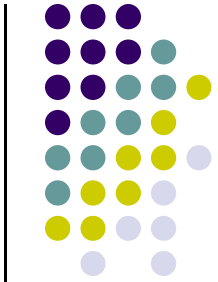
```
A|     // now loop over full data, in bite-sized chunks
B|    for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
C|       // enqueue copies of a in stream0 and stream1
D|        HANDLE_ERROR( cudaMemcpyAsync( dev_a0, host_a+i,   N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );
E|        HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );
F|        // enqueue copies of b in stream0 and stream1
G|        HANDLE_ERROR( cudaMemcpyAsync( dev_b0, host_b+i,   N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );
H|        HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );
I|
J|        // enqueue kernels in stream0 and stream1
K|        kernel<<<(N+255),256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );
L|        kernel<<<(N+255),256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );
M|
N|        // enqueue copies of c from device to locked memory
O|        HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c0,   N * sizeof(int), cudaMemcpyDeviceToHost, stream0 ) );
P|        HANDLE_ERROR( cudaMemcpyAsync( host_c+i+N, dev_c1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1 ) );
Q|    }
```

Replaces Previous Stage 4



Execution timeline of the breadth-first approach
(blue line shows dependency)

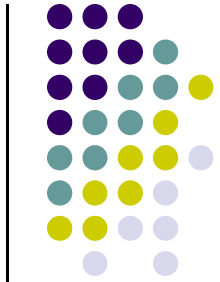
Using Streams, Lessons Learned



- Streams provide a basic mechanism that enables task-level parallelism in CUDA C applications

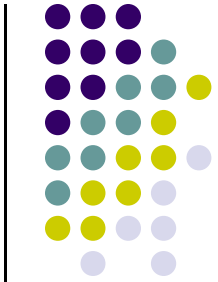
- Two requirements underpin the use of streams in CUDA C
  - `cudaHostAlloc()` should be used to allocate memory on the host so that it can be used in conjunction with a `cudaMemcpyAsync()` non-blocking copy command
    - The use of pinned (page-locked) host memory improves data transfer performance even if you only work with one stream

  - Effective latency hiding of kernel execution with memory copy operations requires a breadth-first approach to enqueuing operations in different streams
    - This is a consequence of the two engine setup associated with a GPU



GPU Computing using `thrust`

3 Ways to Accelerate on GPU



Application

Libraries

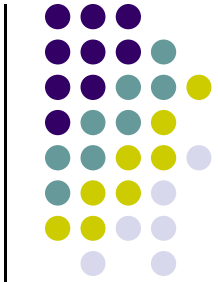
Directives

Programming Languages

Direction of increased performance (and effort)

NVIDIA [C. Woolley]→

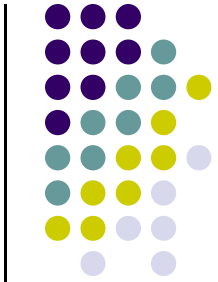
Acknowledgments



- The `thrust` slides include material provided by Nathan Bell of NVIDIA

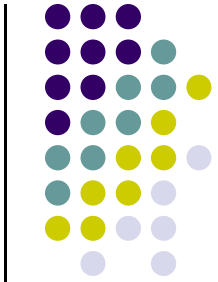
- Any mistakes in these slides belong to me

Design Philosophy, `thrust`



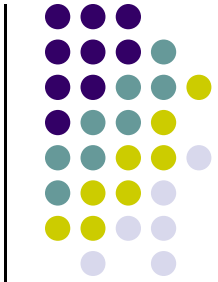
- Increase programmer productivity
  - Build complex applications quickly
- Encourage generic programming
  - Leverage parallel primitives
- Should run fast
  - Efficient mapping to hardware

What is `thrust`?



- A template library for CUDA
  - Mimics the C++ STL
- Containers
  - On host and device
- Algorithms
  - Sorting, reduction, scan, etc.

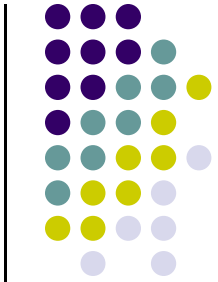
What is `thrust`? [Cntd.]



- `thrust` is a header library – all the functionality is accessed by `#include`-ing the appropriate `thrust` header file
- Program is compiled with `nvcc` as per usual, no special tools are required
- Lots of C++ syntax, related to high-level host-side code that you write
  - The concept of execution configuration, shared memory, etc. : all gone

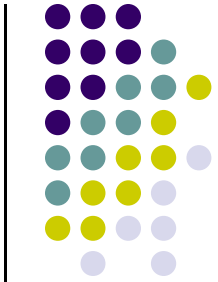
NVIDIA [N. Bell]→

Why Should One Use `thrust`?

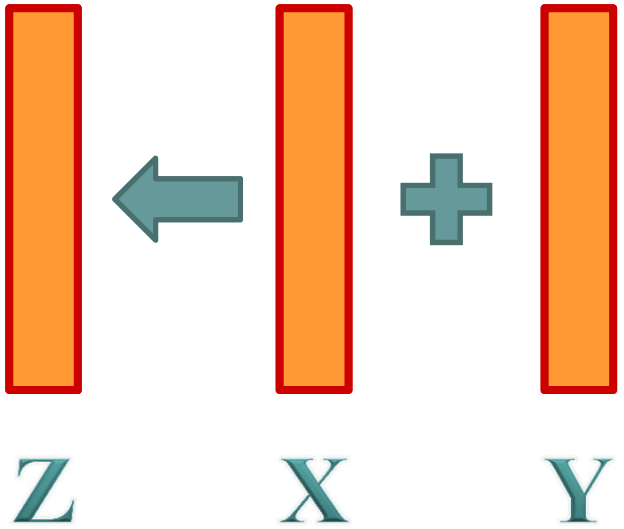


- Extensively tested
  - 600+ unit tests
- Open Source
  - Permissive License (Apache v2)
- Active community

Example: Vector Addition

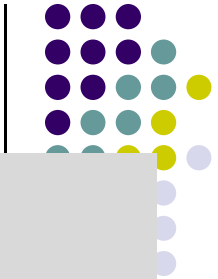


```
for (int i = 0; i < N; i++)
    Z[i] = X[i] + Y[i];
```



NVIDIA [N. Bell]→

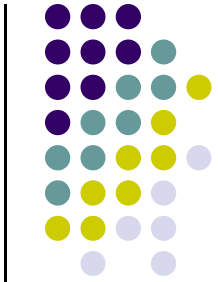
Example, Vector Addition



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

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

  X[0] = 10; X[1] = 20; X[2] = 30;
  Y[0] = 15; Y[1] = 35; Y[2] = 10;

  thrust::transform(X.begin(), X.end(),
                    Y.begin(),
                    Z.begin(),
                    thrust::plus<float>());

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

  return 0;
}
```


NVIDIA [N. Bell]→

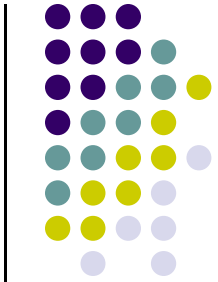
Example, Vector Addition



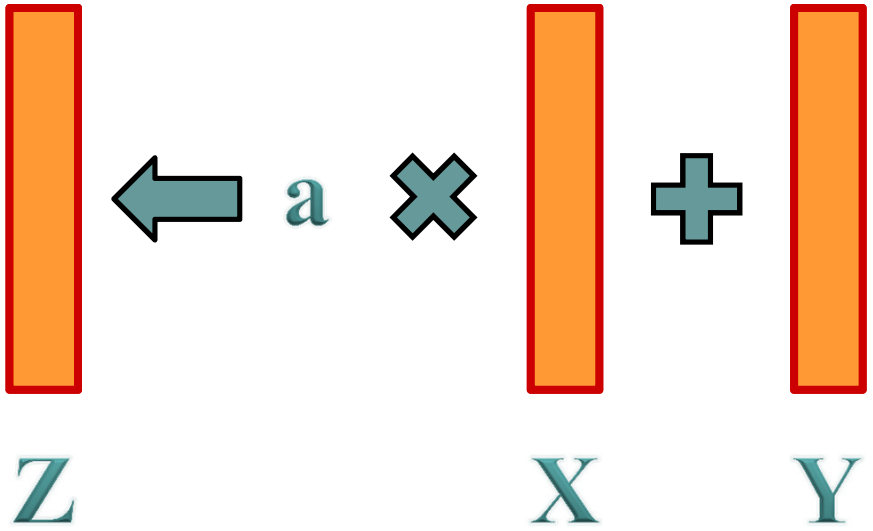
```
[negrut@euler01 CodeBits]$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2011 NVIDIA Corporation
Built on Thu_Jan_12_14:41:45_PST_2012
Cuda compilation tools, release 4.1, V0.2.1221
[negrut@euler01 CodeBits]$ nvcc -O2 exThrust.cu -o exThrust.exe
[negrut@euler01 CodeBits]$ ./exThrust.exe
Z[0] = 25
Z[1] = 55
Z[2] = 40
[negrut@euler01 CodeBits]$
```

- Note: file extension should be **`.cu`**

Example: SAXPY



```
for (int i = 0; i < N; i++)
    Z[i] = a * X[i] + Y[i];
```



NVIDIA [N. Bell]→

SAXPY

functor

```
struct saxpy
{
  float a;

  saxpy(float a) : a(a) {}

  __host__ __device__
  float operator()(float x, float y)
  {
    return a * x + y;
  }
};
```

state

constructor

call operator

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

  X[0] = 10; X[1] = 20; X[2] = 30;
  Y[0] = 15; Y[1] = 35; Y[2] = 10;

  float a = 2.0f;

  thrust::transform(X.begin(), X.end(),
                    Y.begin(),
                    Z.begin(),
                    saxpy(a));

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

  return 0;
}
```

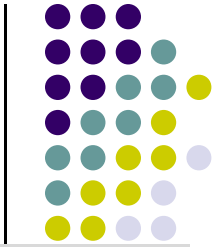

NVIDIA [N. Bell]→

SAXPY

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

using namespace thrust::placeholders;

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

  X[0] = 10; X[1] = 20; X[2] = 30;
  Y[0] = 15; Y[1] = 35; Y[2] = 10;

  float a = 2.0f;

  thrust::transform(X.begin(), X.end(),
                    Y.begin(),
                    Z.begin(),
                    a * _1 + _2);

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

  return 0;
}
```

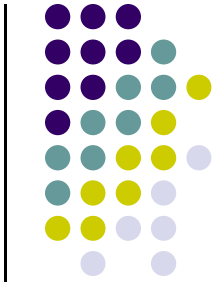
NVIDIA [N. Bell]→

Diving In



```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.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;
}
```

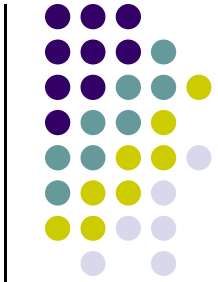
Containers



- Concise and readable code
  - Avoids common memory management errors
    - e.g.: Vectors automatically release memory when they go out of scope

```
// 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;

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

// read device values from the host
std::cout << "sum: " << d_vec[0] + d_vec[1] << std::endl;
```

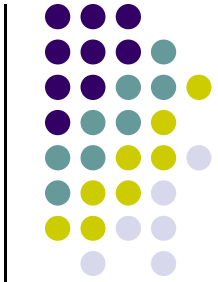
Containers



- Compatible with STL containers

```
// 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 using vector constructor
thrust::device_vector<int> d_vec(h_list.begin(), h_list.end());
```

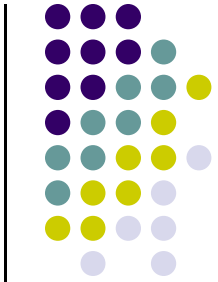
Namespaces



- Avoid name collisions

```
// allocate host memory
thrust::host_vector<int> h_vec(10);

// call STL sort
std::sort(h_vec.begin(), h_vec.end());

// call Thrust sort
thrust::sort(h_vec.begin(), h_vec.end());

// for brevity
using namespace thrust;

// without namespace
int sum = reduce(h_vec.begin(), h_vec.end());
```

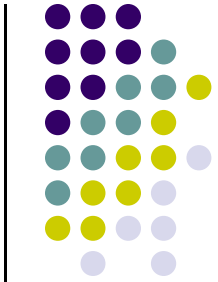
Iterators



- A pair of iterators defines a “range”

```
// allocate device memory
device_vector<int> d_vec(10);

// declare iterator variables
device_vector<int>::iterator begin  = d_vec.begin();
device_vector<int>::iterator end    = d_vec.end();
device_vector<int>::iterator middle = begin + d_vec.size()/2;
// sum first and second halves
int sum_half1 = reduce(begin, middle);
int sum_half2 = reduce(middle, end);

// empty range
int empty = reduce(begin, begin);
```

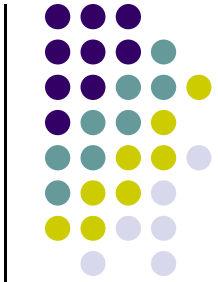
Iterators



- Iterators act like pointers

```
// declare iterator variables
device_vector<int>::iterator begin = d_vec.begin();
device_vector<int>::iterator end   = d_vec.end();

// pointer arithmetic
begin++;

// dereference device iterators from the host
int a = *begin;
int b = begin[3];

// compute size of range [begin,end)
int size = end - begin;
```

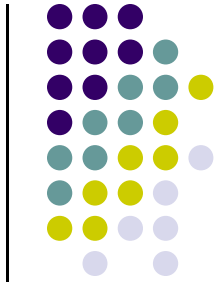
Iterators



- Encode memory location
  - Automatic algorithm selection

```
// initialize random values on host
host_vector<int> h_vec(100);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

// copy values to device
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

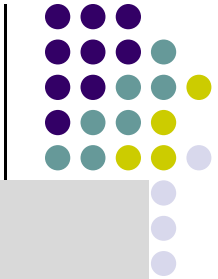


- Elementwise operations
  - `for_each`, `transform`, `gather`, `scatter` …
- Reductions
  - `reduce`, `inner_product`, `reduce_by_key` …
- Prefix-Sums
  - `inclusive_scan`, `inclusive_scan_by_key` …
- Sorting
  - `sort`, `stable_sort`, `sort_by_key` …

Thrust Example: Sort

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

int main(void) {
  // generate 16M random numbers on the host
  thrust::host_vector<int> h_vec(1 << 24);
  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 (805 Mkeys/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;
}
```

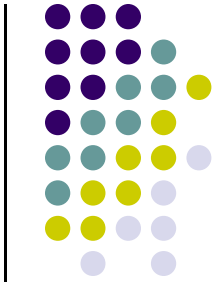
Maximum Value



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

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

  X[0] = 10; X[1] = 30; X[2] = 20;

  float init = 0.0f;

  float result = thrust::reduce(X.begin(), X.end(),
                                init,
                                thrust::maximum<float>());

  std::cout << "maximum is " << result << "\n";

  return 0;
}
```


NVIDIA [N. Bell]→

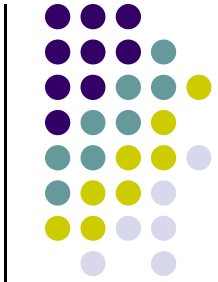
Algorithms



- Process one or more ranges

```
// copy values to device
device_vector<int> A(10);
device_vector<int> B(10);
device_vector<int> C(10);

// sort A in-place
sort(A.begin(), A.end());

// copy A -> B
copy(A.begin(), A.end(), B.begin());

// transform A + B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus<int>());
```

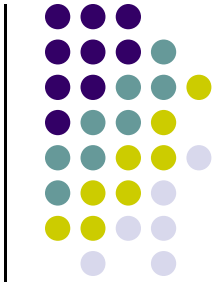
Algorithms



- Standard operators

```
// allocate memory
device_vector<int>  A(10);
device_vector<int>  B(10);
device_vector<int>  C(10);

// transform A + B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), plus<int>());

// transform A - B -> C
transform(A.begin(), A.end(), B.begin(), C.begin(), minus<int>());

// multiply reduction
int product = reduce(A.begin(), A.end(), 1, multiplies<int>());
```

Algorithms

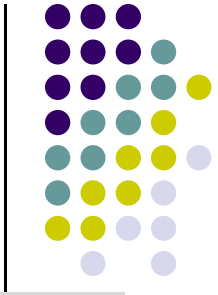


- Standard data types

```
// allocate device memory
device_vector<int>   i_vec = ...
device_vector<float> f_vec = ...

// sum of integers
int i_sum = reduce(i_vec.begin(), i_vec.end());

// sum of floats
float f_sum = reduce(f_vec.begin(), f_vec.end());
```

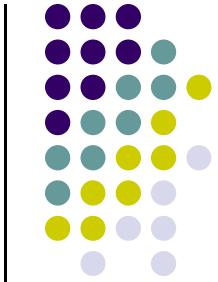
NVIDIA [N. Bell]→

Custom Types & Operators



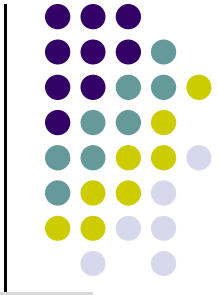
```
struct negate_float2
{
    __host__ __device__
    float2 operator()(float2 a)
    {
        return make_float2(-a.x, -a.y);
    }
};

// declare storage
device_vector<float2> input  = ...
device_vector<float2> output = ...

// create function object or 'functor'
negate_float2 func;

// negate vectors
transform(input.begin(), input.end(), output.begin(), func);
```

Custom Types & Operators



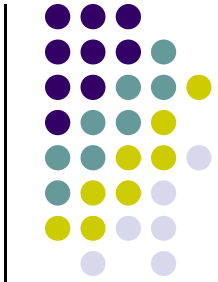
```
// compare x component of two float2 structures
struct compare_float2
{
    __host__ __device__
    bool operator()(float2 a, float2 b)
    {
        return a.x < b.x;
    }
};

// declare storage
device_vector<float2> vec = ...

// create comparison functor
compare_float2 comp;

// sort elements by x component
sort(vec.begin(), vec.end(), comp);
```

Custom Types & Operators



```
// return true if x is greater than threshold
struct is_greater_than
{
   int threshold;

   is_greater_than(int t) { threshold = t; }

   __host__ __device__
   bool operator()(int x) { return x > threshold; }
};

device_vector<int> vec = ...

// create predicate functor (returns true for x > 10)
is_greater_than pred(10);

// count number of values > 10
int result = count_if(vec.begin(), vec.end(), pred);
```

Interoperability

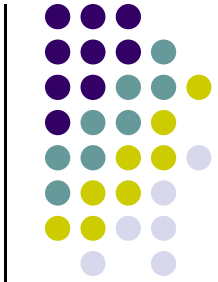


- Convert iterators to raw pointers

```
// 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);

// use ptr in a CUDA API function
cudaMemcpyAsync(ptr, ... );
```


NVIDIA [N. Bell]→

Interoperability



- Wrap raw pointers with `device_ptr`

```
// raw pointer to device memory
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof(int));

// wrap raw pointer with a device_ptr
thrust::device_ptr<int> dev_ptr(raw_ptr);

// use device_ptr in thrust algorithms
thrust::fill(dev_ptr, dev_ptr + N, (int) 0);

// access device memory through device_ptr
dev_ptr[0] = 1;

// free memory
cudaFree(raw_ptr);
```


NVIDIA [N. Bell]→