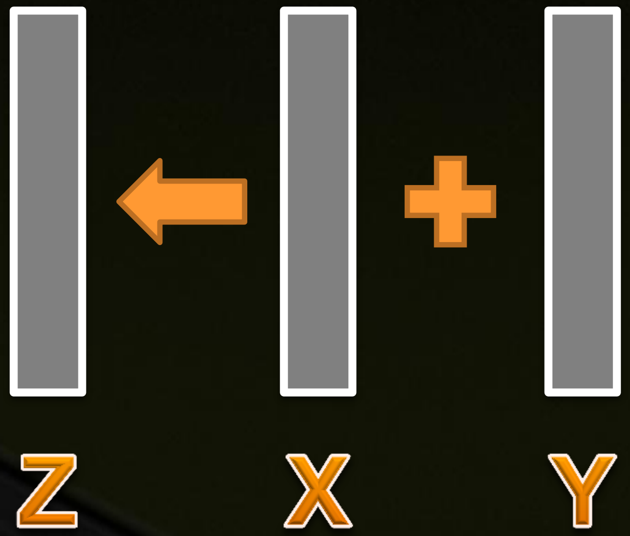
# Rapid Problem Solving Using Thrust

# Vector Addition

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



# Vector Addition

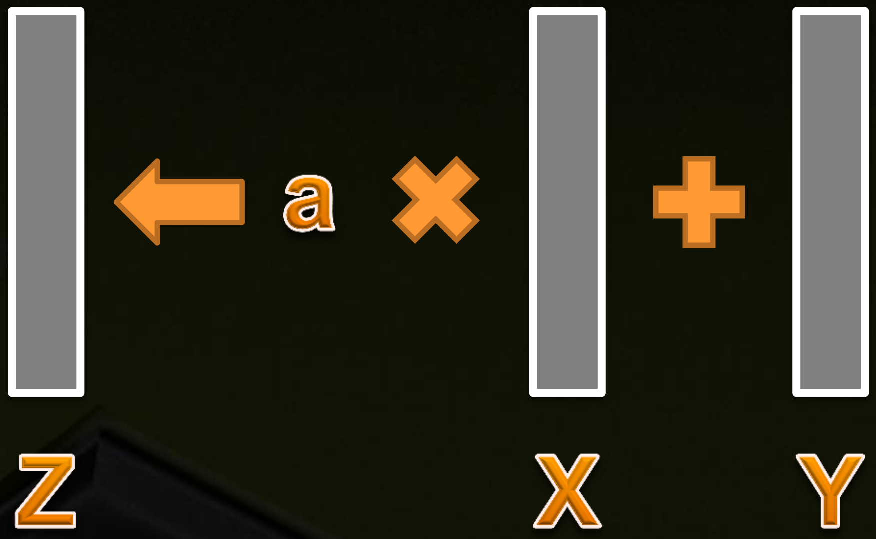
```
#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <iostream>

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;
}
```

# Vector Addition

```
ProblemSolving$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2011 NVIDIA Corporation
Built on Thu_May_12_11:09:45_PDT_2011
Cuda compilation tools, release 4.0, V0.2.1221
ProblemSolving$ nvcc -O2 ex01_vector_addition.cu -o ex01_vector_addition
ProblemSolving$ ./ex01_vector_addition
Z[0] = 25
Z[1] = 55
Z[2] = 40
```

# SAXPY

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



# SAXPY

```
struct saxpy
{
  float a;

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

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

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;
}
```

functor

state

constructor

call operator

# 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;
}
```

New Feature in Thrust v1.5 (CUDA 4.1)

# General Transformations

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],...);
```

# General Transformations


Multiple Sequences

Sequence of Tuples

# General Transformations

```
#include <thrust/iterator/zip_iterator.h>

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);

  X[0] = 10; X[1] = 20; X[2] = 30;
  Y[0] = 15; Y[1] = 35; Y[2] = 10;
  Z[0] = 20; Z[1] = 30; Z[2] = 25;

  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;
}
```

# Sum

```
#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 result = thrust::reduce(X.begin(), X.end());

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

  return 0;
}
```

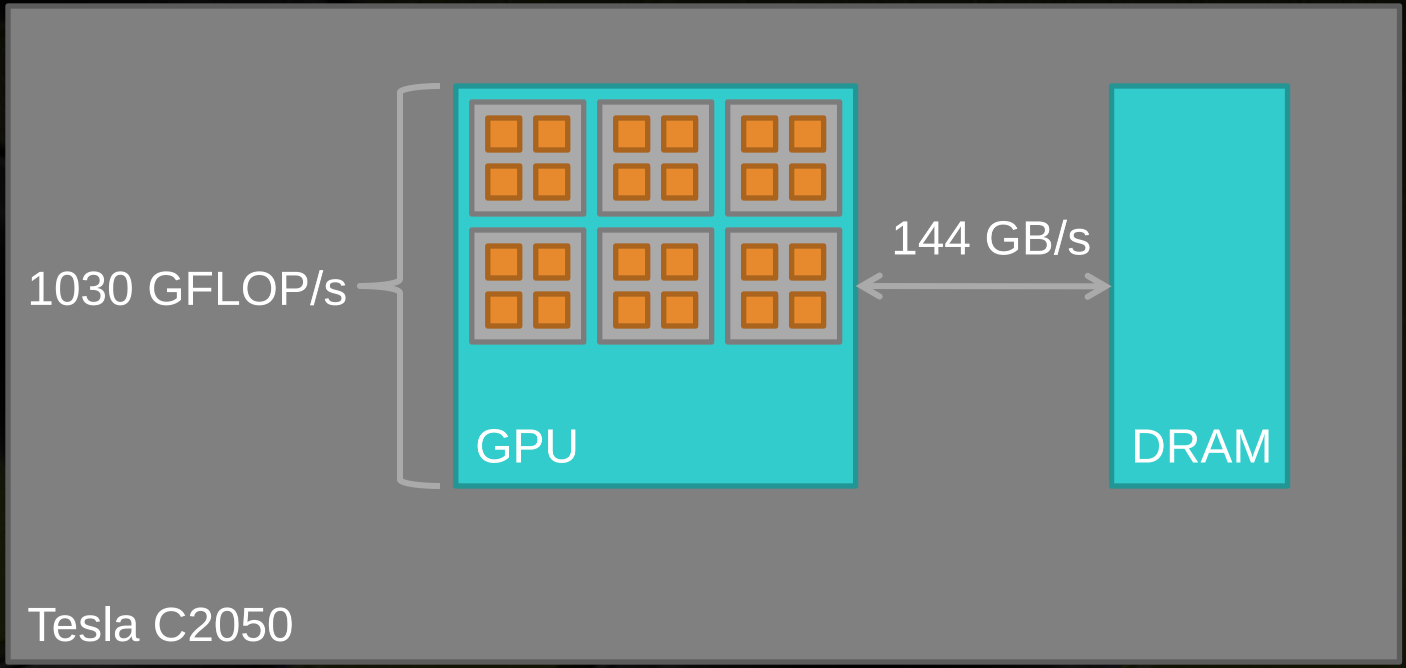
# Maximum Value

```
#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;
}
```

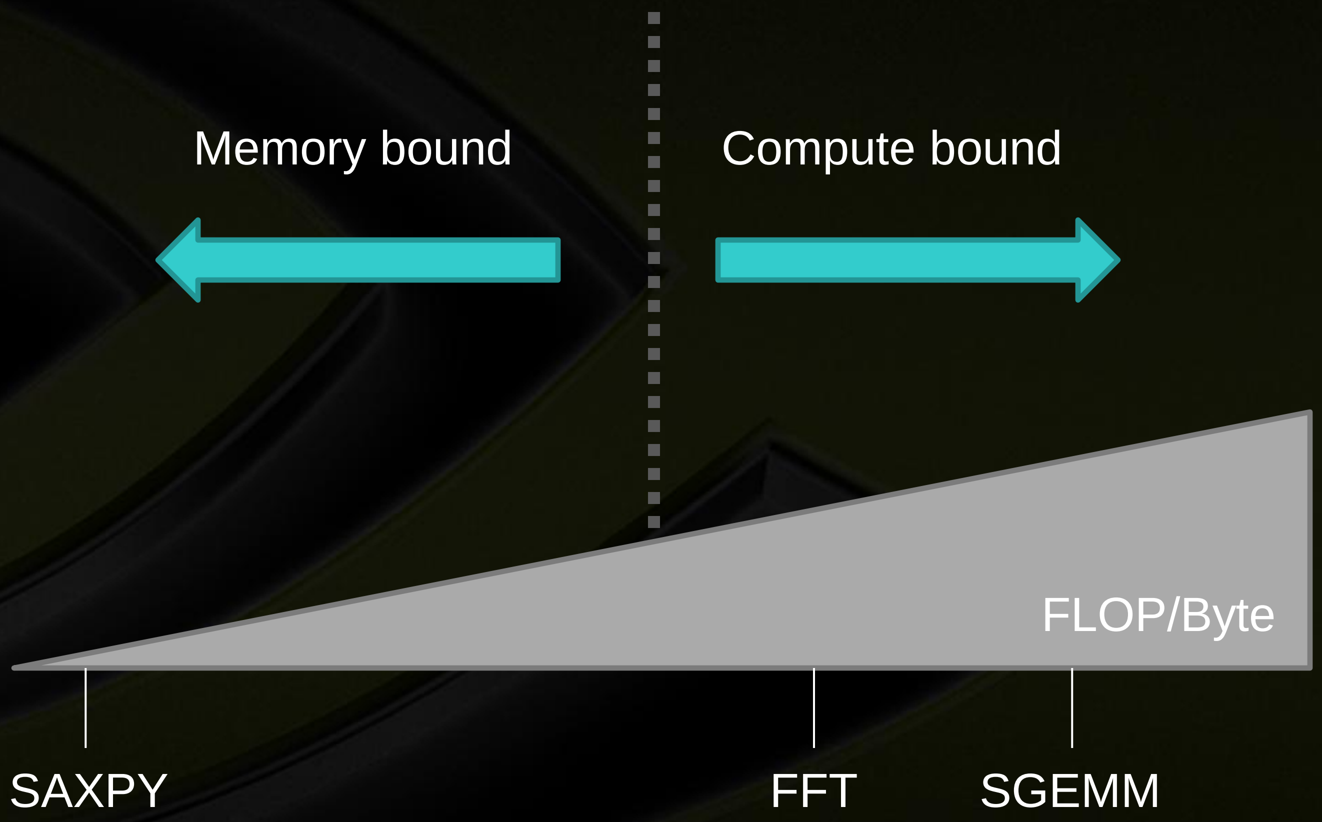
# Maximum Index

```
typedef thrust::tuple<int,int> Tuple;

struct max_index
{
  __host__ __device__
  Tuple operator()(Tuple a, Tuple b)
  {
    if (thrust::get<0>(a) > thrust::get<0>(b))
      return a;
    else
      return b;
  }
};

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

  X[0] = 10; X[1] = 30; X[2] = 20;   // values
  Y[0] =  0; Y[1] =  1; Y[2] =  2;   // indices

  Tuple init(X[0],Y[0]);

  Tuple result = thrust::reduce
    (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y.begin())),
     thrust::make_zip_iterator(thrust::make_tuple(X.end(),   Y.end())),
     init,
     max_index());

  int value, index;  thrust::tie(value,index) = result;

  std::cout << "maximum value is " << value << " at index " << index << "\n";

  return 0;
}
```

# Performance Considerations



# Arithmetic Intensity



# Arithmetic Intensity

| Kernel | FLOP/Byte** |
|---|---|
| Vector Addition | 1 : 12 |
| SAXPY | 2 : 12 |
| Ternary Transformation | 5 : 20 |
| Sum | 1 : 4 |
| Max Index | 1 : 12 |

| Kernel | FLOP/Byte |
|---|---|
| GeForce GTX 280 | ~7.0 : 1 |
| GeForce GTX 480 | ~7.6 : 1 |
| Tesla C870 | ~6.7 : 1 |
| Tesla C1060 | ~9.1 : 1 |
| Tesla C2050 | ~7.1 : 1 |

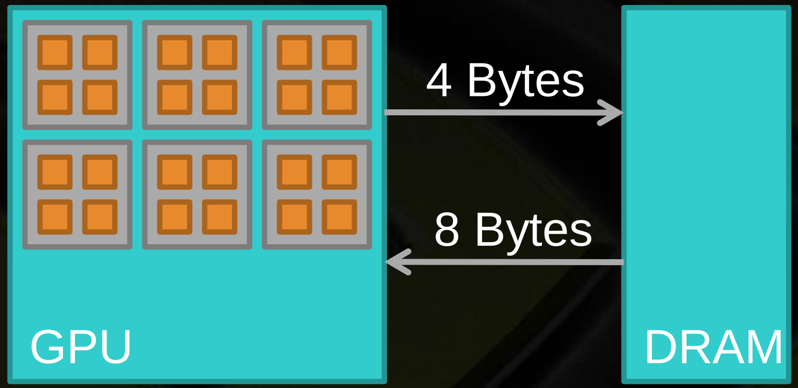
** excludes indexing overhead

# Maximum Index (Optimized)

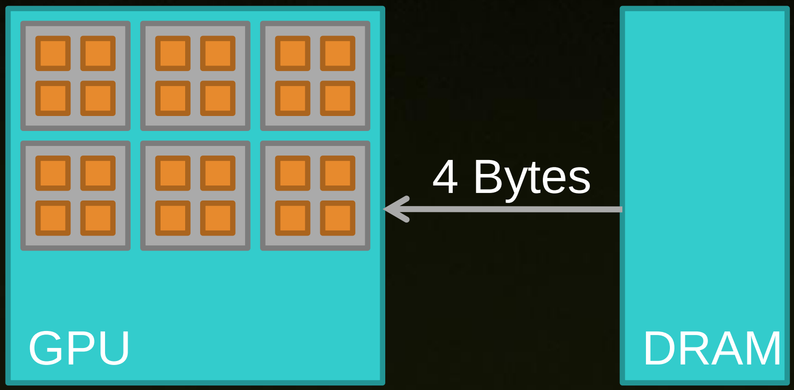
```
typedef thrust::tuple<int,int> Tuple;

struct max_index
{
  __host__ __device__
  Tuple operator()(Tuple a, Tuple b)
  {
    if (thrust::get<0>(a) > thrust::get<0>(b))
      return a;
    else
      return b;
  }
};

int main(void)
{
  thrust::device_vector<int>     X(3);
  thrust::counting_iterator<int> Y(0);

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

  Tuple init(X[0],Y[0]);

  Tuple result = thrust::reduce
    (thrust::make_zip_iterator(thrust::make_tuple(X.begin(), Y)),
     thrust::make_zip_iterator(thrust::make_tuple(X.end(),   Y + X.size())),
     init,
     max_index());

  int value, index;  thrust::tie(value,index) = result;

  std::cout << "maximum value is " << value << " at index " << index << "\n";

  return 0;
}
```

# Maximum Index (Optimized)

Original Implementation




Optimized Implementation



# 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]);
}
```

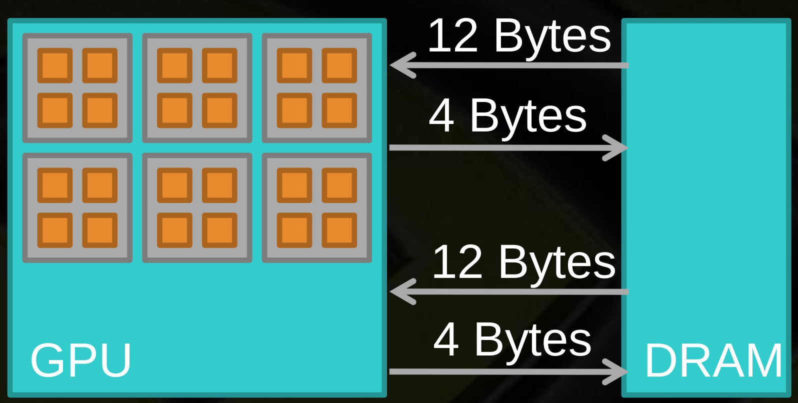
Loop Fusion

# Fusing Transformations

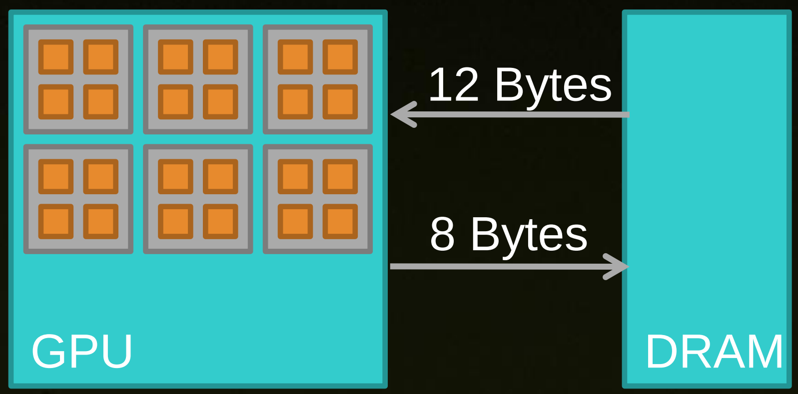
```
typedef thrust::tuple<float,float>       Tuple2;
typedef thrust::tuple<float,float,float> Tuple3;

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);

  X[0] = 10; X[1] = 20; X[2] = 30;
  Y[0] = 15; Y[1] = 35; Y[2] = 10;
  Z[0] = 20; Z[1] = 30; Z[2] = 25;

  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



# 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]);
```

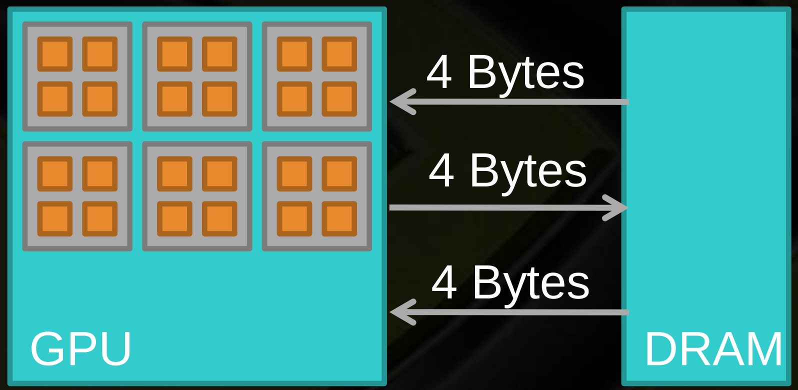
Loop Fusion

# Fusing Transformations

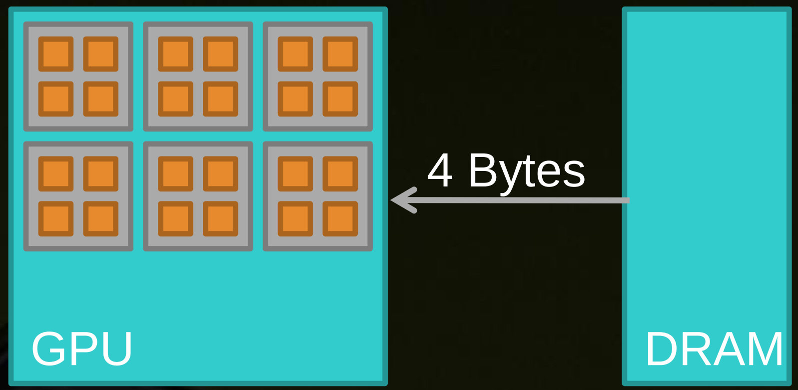
```
#include <thrust/device_vector.h>
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>
#include <iostream>

using namespace thrust::placeholders;

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

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

  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;
}
```

# Fusing Transformations

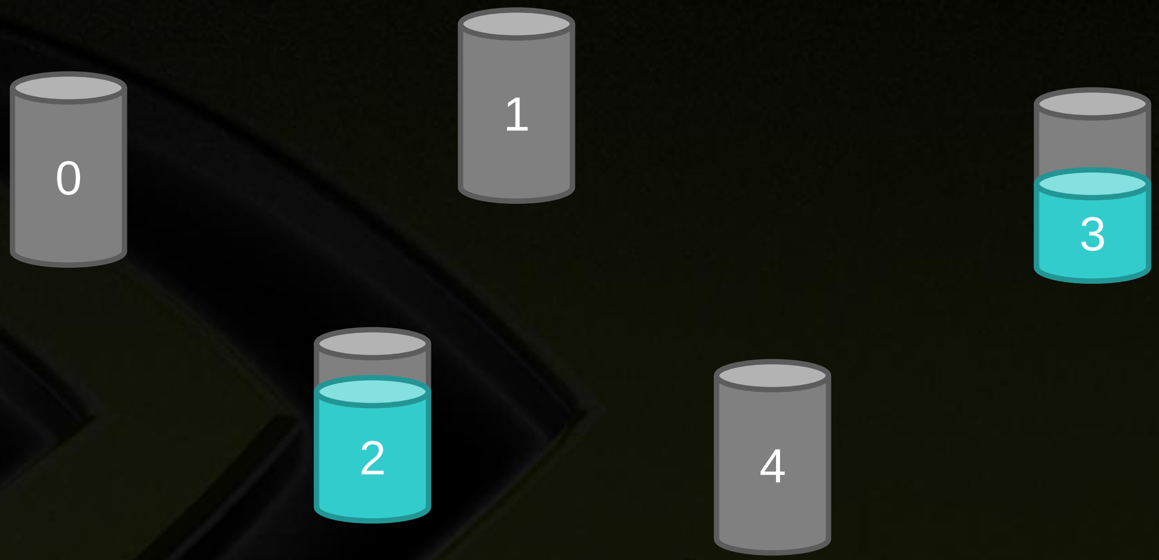
Original Implementation




Optimized Implementation



# Example: Processing Rainfall Data




```
day         [0  0  1  2  5  5  6  6  7   8  ... ]
site        [2  3  0  1  1  2  0  1  2   1  ... ]
measurement [9  5  6  3  3  8  2  6  5  10  ... ]
```

Notes
1) Time series sorted by day
2) Measurements of zero are excluded from the time series

# Example: Processing Rainfall Data

- Total rainfall at a given site
- Total rainfall between given days
- Number of days with any rainfall
- Total rainfall on each day

# Total Rainfall at a Given Site

```
struct one_site_measurement
{
  int site;

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

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

template <typename Vector>
int compute_total_rainfall_at_one_site(int i, 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(i),
     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)
{
  typedef typename Vector::iterator Iterator;

  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);
}
```

```
                        lower_bound( ... , 2)       upper_bound( ... , 6)
                                 ↓                              ↓
day          [0    0    1    2    5    5    6    6    7    8    ... ]
measurement  [9    5    6    3    3    8    2    6    5   10    ... ]
```

# Number of Days with Any Rainfall

```
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;
}
```

day [0 = 0 ≠ 1 ≠ 2 ≠ 5 = 5 ≠ 6 = 6 ≠ 7 ≠ 8 ... ]

[0 + 1 + 1 + 1 + 0 + 1 + 0 + 1 + 1 ... ] + 1

# Total Rainfall on Each Day

```
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);

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

  thrust::reduce_by_key(day.begin(), day.end(),
                        measurement.begin(),
                        day_output.begin(),
                        measurement_output.begin());
}
```

```
day          [0 = 0   1   2   5 = 5   6 = 6   7   8   ... ]
measurement  [9 + 5   6   3   3 + 8   2 + 6   5  10   ... ]
```

```
output_day          [0   1   2   5   6   7   8   ... ]
output_measurement  [14  6   3  11   8   5  10   ... ]
```

# Homework

- Number of days where rainfall exceeded 5
  - Use `count_if` and a placeholder

- Total Rainfall at Each Site
  - Use `sort_by_key` and `reduce_by_key`

# Interoperability

- Convert iterators to raw pointers

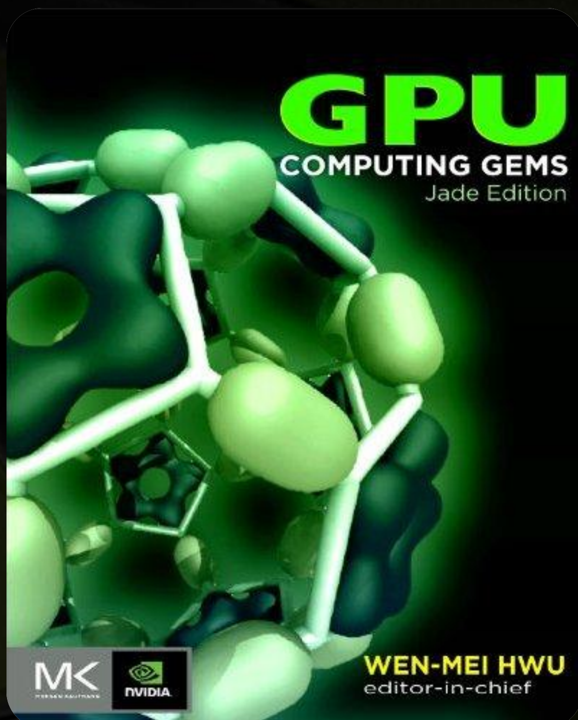
```
// allocate device vector
thrust::device_vector<int> d_vec(4);

// 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, ... );
```

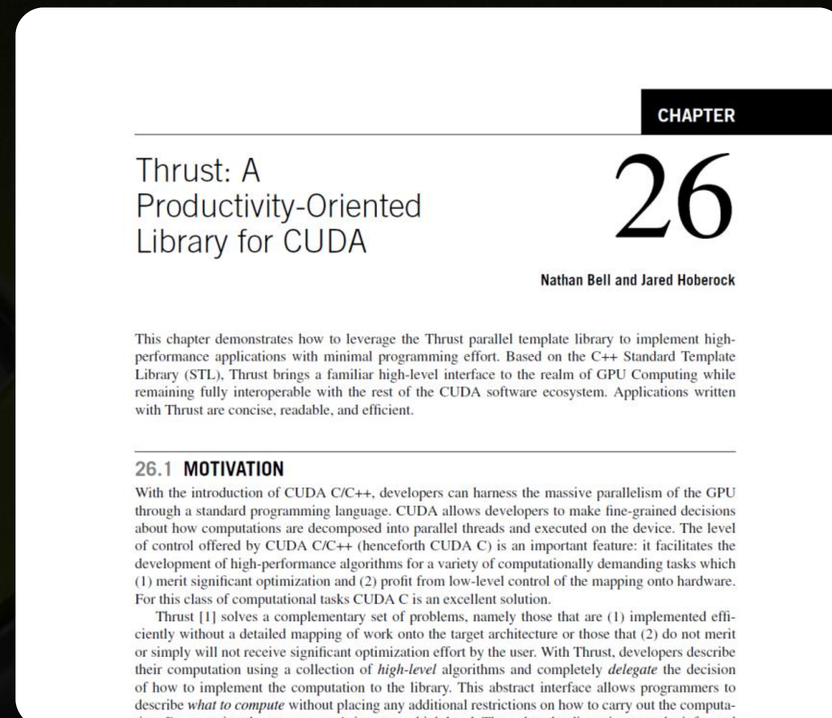
# Interoperability

- 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);
```

# Thrust in GPU Computing Gems





CHAPTER

# Thrust: A Productivity-Oriented Library for CUDA

26

Nathan Bell and Jared Hoberock

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

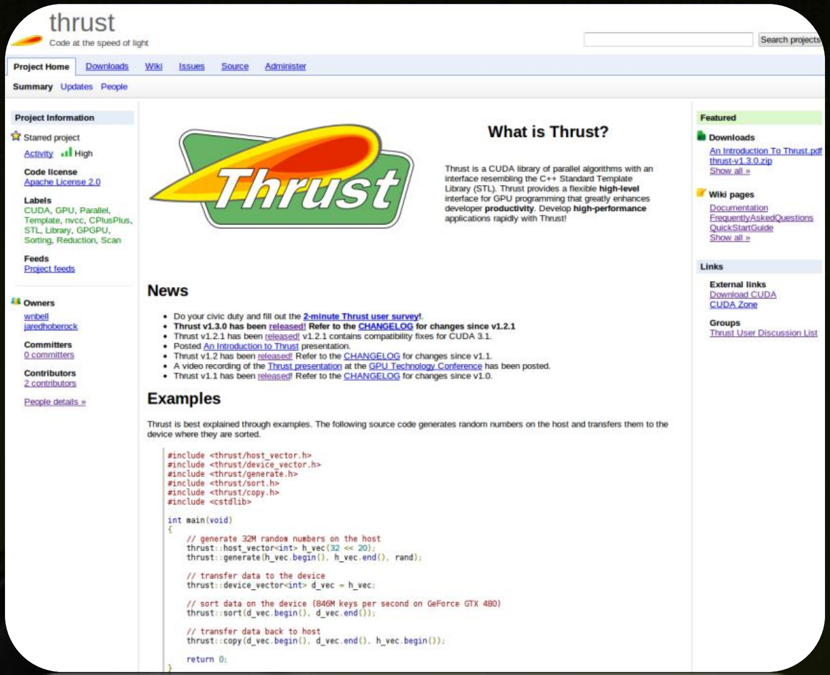
With the introduction of CUDA C/C++, developers can harness the massive parallelism of the GPU through a standard programming language. CUDA allows developers to make fine-grained decisions about how computations are decomposed into parallel threads and executed on the device. The level of control offered 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 capturing the programmer's intent at a high level, Thrust has the discretion to make informed

PDF available at http://goo.gl/adj9S

# Thrust on Google Code

- Quick Start Guide
- Examples
- News
- Documentation
- Mailing List (thrust-users)



# Register for the Next GTC Express

*Introduction to Parallel Nsight and Features Preview for Version 2.1*

Shane Evans, Product Manager, NVIDIA

Wednesday, October 12, 2011, 9:00 AM PDT

Parallel Nsight is NVIDIA's powerful solution for GPGPU and graphics analysis and debugging. By attending the webinar you'll

- Take a deep dive into prominent features
- Get tremendous visibility into thread activity and memory
- Get help optimizing kernel code, such as Branching Efficiency, Branch Statistics, Achieved FLOPs, and more
- Get a sneak preview of upcoming features of version 2.1

**Register at www.gputechconf.com**