Title: PISTON

Author(s): Christopher Sewell
Li-ta Lo
James Ahrens

Intended for: Guest Lectures at the University of Oregon, presented remotely, January 2014
PISTON

Chris Sewell
Li-ta Lo
James Ahrens
Los Alamos National Laboratory

Lectures for Hank Child’s class on many-core visualization at the University of Oregon, January 2014
Outline

- Practical Introduction to High-Level Data Parallelism using Thrust and PISTON
- Research Talk Overview of PISTON and PINION
- Additional Tutorial Examples
- Additional Details on Research Results
PRACTICAL INTRODUCTION TO HIGH-LEVEL DATA PARALLELISM USING THRUST AND PISTON
Advantages of High-Level Parallel Programming

- Supercomputer Hardware Advances Everyday
  - Higher and higher parallelism
  - Optimizations tailored to a certain architecture will be obsolete when you implement it

- Parallel Programming APIs Come and Go
  - Nobody programs with shaders for GPGPU anymore
  - Will this also happen to OpenCL, CUDA, etc. in the future?

- High-Level Parallelism
  - Will not change over time
NVIDIA’s Thrust Library

- Thrust is an open-source C++ template library developed by NVIDIA

- It allows the user to write CUDA programs using an STL-like interface, without having to know CUDA-specific syntax or functions

- In addition to CUDA, it has backends for OpenMP and Intel TBB, and can be extended to support additional backends

- It implements many data-parallel primitives, with user-defined functors

- It provides thrust::host_vector and thrust::device_vector, simplifying memory management and data transfer between the host and device

---

Sample Thrust code to compute vector norm

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

// square<T> computes the square of a number f(x) -> x*x
template <typename T>
struct square
{
  __host__ __device__
  T operator()(const T& x) const {
    return x * x;
  }
};

int main(void)
{
  // initialize host array
  float x[4] = {1.0, 2.0, 3.0, 4.0};

  // transfer to device
  thrust::device_vector<float> d_x(x, x + 4);

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

  // compute norm
  float norm = std::sqrt(thrust::transform_reduce(d_x.begin(),
                                              d_x.end(), unary_op, init, binary_op));

  std::cout << norm << std::endl;
  return 0;
}
How PISTON/PINION Leverage Thrust

- Thrust provides:
  - An STL-like interface for memory management (host/device vectors) and data-parallel algorithms
  - Backend implementations of the data-parallel algorithms for CUDA, as well as slightly less-developed implementations for OpenMP and TBB

- PISTON/PINION intend to provide:
  - A library of visualization and analysis operators implemented using Thrust
  - A data model for simulation meshes (e.g., VTK structured grids, unstructured grids, AMR)
  - Simulation operators (e.g., advection, interface reconstruction, etc.)

- PISTON/PINION intend to enhance:
  - Non-CUDA backends (e.g., OpenCL prototype, optimize OpenMP for Xeon Phi, etc.)
  - Interface to support distributed memory operations
Sample Thrust Program

```c
#include <thrust/device_vector.h>
#include <thrust/iterator/constant_iterator.h>
#include <thrust/scalar.h>
#include <thrust/sort.h>

struct saapy_functor
{
  const float a;
  saapy_functor(float _a : a) {}  
};

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

int main(int)
{
  // Declare host and device vectors
  thrust::host_vector<int> h1(5), h2(5), h3;
  thrust::device_vector<int> d1, d2, d3;

  // Initialize h1 with [2, 2, 2, 2, 2] and h2 with [1, 1, 1, 1, 1]
  thrust::fill(h1.begin(), h1.end(), 2);
  thrust::fill(h2.begin(), h2.end(), 1);

  // Copy data from host vectors to device vectors
  d1 = h1;  d2 = h2;

  // Add D1 and D2 and store in D3
  thrust::transform(d1.begin(), d1.end(), d2.begin(), d3.begin(), thrust::plus<int>());

  // Test operator
  HM = d1;  std::cout << HM[i];  for (int i = 0; i < HM.size(); i++) std::cout << HM[i] << " " " ; std::cout << std::endl;

  // Perform inclusive scan (prefix sum) on D1 and store in D2
  thrust::inclusive_scan(d1.begin(), d1.end(), d2.begin(), thrust::plus<int>());

  // Test inclusive scan
  HM = d2;  std::cout << HM[i] << " " " ; for (int i = 0; i < HM.size(); i++) std::cout << HM[i] << " " " ; std::cout << std::endl;

  // Use a permutation iterator to gather elements from D1 based on specified indices in D2, storing in HM
  HM = d2;  std::cout << HM[i];  for (int i = 0; i < HM.size(); i++) std::cout << HM[i] << " " " ; std::cout << std::endl;

  // Compute a hash (4x + y) where x is D1, y is [5, 9, 13, 17] and 4 is 2, storing result in D3
  thrust::transform(d1.begin(), d1.end(), thrust::make_constant_iterator(2), d3.begin(), saapy_functor(2));

  // Test hash
  HM = d3;  std::cout << HM[i] << " " " ; for (int i = 0; i < HM.size(); i++) std::cout << HM[i] << " " " ; std::cout << std::endl;

  // Copy HM to D1, fill D3 with [0, 1, 2, 3, 4], and sort D1, D2, and D3 based on values in D1
  HM = d3;  std::cout << HM[i];  for (int i = 0; i < HM.size(); i++) std::cout << HM[i] << " " " ; std::cout << std::endl;

  // Test sort by key
  thrust::sort_by_key(d1.begin(), d1.end(), thrust::make_zip_iterator(thrust::make_tuple(thrust::make_counting_iterator(0), d2.begin(), d3.begin())));

  // Test sort by key
  thrust::sort_by_key(d1.begin(), d1.end(), thrust::make_zip_iterator(thrust::make_counting_iterator(0), d2.begin(), d3.begin()));

  // Compute sum of elements in D2; sum = 55
  HM = d2;  std::cout << HM[i];  for (int i = 0; i < HM.size(); i++) std::cout << HM[i] << " " " ; std::cout << std::endl;

  // Test reduce
  HM = d2;  std::cout << HM[i];  for (int i = 0; i < HM.size(); i++) std::cout << HM[i] << " " " ; std::cout << std::endl;

  return 0;
}
```

> nvcc examples.cu -o examples
>
> ./examples
#include <thrust/device_vector.h>
#include <thrust/iterator/constant_iterator.h>
#include <thrust/scan.h>
#include <thrust/sort.h>

struct saxpy_functor
{
    const float a;
    saxpy_functor(float _a) : a(_a) {}

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

int main(void)
{
    // Declare host and device vectors
    thrust::host_vector<int> H1(5), H2(5), H3;
    thrust::device_vector<int> D1, D2, D3;

    // Initialize H1 with [2,2,2,2,2] and H2 with [0,1,2,3,4]
    thrust::fill(H1.begin(), H1.end(), 2);
    thrust::sequence(H2.begin(), H2.end(), 0);

    // Copy data from host vectors to device vectors
    D1 = H1; D2 = H2;

    // Add D1 and D2 and store in D1
    thrust::transform(D1.begin(), D1.end(), D2.begin(), D1.begin(), thrust::plus<int*>());

    // H1 = D1 = [2,3,4,5,6]
    H1 = D1; std::cout << 'H1: ';
    for (int i=0; i<H1.size(); i++) std::cout << ' ' << H1[i] << ' '; std::cout << std::endl;

    // Perform inclusive scan (prefix sum) on D1 and store in D2
    thrust::inclusive_scan(D1.begin(), D1.end(), D2.begin());
```cpp
// H2 = D2 = [2,5,9,14,20]
H2 = D2; std::cout << 'H2: ' + std::to_string(H2.size()) + ', std::cout << H2[i] << ' ', std::cout << std::endl;

// Use a permutation iterator to gather elements from D1 based on specified indices in D2, storing in H1
D2[0] = 1; D2[1] = 4; D2[2] = 2; D2[3] = 0; D2[4] = 3;
thrust::copy(thrust::make_permutation_iterator(D1.begin(), D2.begin()), thrust::make_permutation_iterator(D1.begin(), D2.end()), H1.begin());

// H1 = [3, 6, 4, 2, 5]
for (int i=0; i<H1.size(); i++) std::cout << H1[i] << ' ', std::cout << std::endl;

// Compute saxby (A*x + y) where x is D1, y is [3,3,3,3,3] and A is 2, storing result in D2
thrust::transform(D1.begin(), D1.end(), thrust::make_constant_iterator(3), D2.begin(), saxpy functor(2));

// H2 = D2 = [7, 9, 11, 13, 15]
H2 = D2; std::cout << 'H2: ' + std::to_string(H2.size()) + ', std::cout << H2[i] << ' ', std::cout << std::endl;

// Copy H1 to D1, fill D3 with [0,1,2,3,4], and sort D1, D2, and D3 based on values in D1
D1 = H1;
D3.resize(5); thrust::copy(thrust::make_counting_iterator(0), thrust::make_counting_iterator(0)+5, D3.begin());
thrust::sort_by_key(D1.begin(), D1.end(), thrust::make_zip_iterator(thrust::make_tuple(D2.begin(), D3.begin())));

// H1 = D1 = [2, 3, 4, 5, 6]; H2 = D2 = [13, 7, 11, 15, 9]; H3 = D3 = [3, 0, 2, 4, 1]
H1 = D1; std::cout << 'H1: ' + std::to_string(H1.size()) + ', std::cout << H1[i] << ' ', std::cout << std::endl;
H2 = D2; std::cout << 'H2: ' + std::to_string(H2.size()) + ', std::cout << H2[i] << ' ', std::cout << std::endl;
H3 = D3; std::cout << 'H3: ' + std::to_string(H3.size()) + ', std::cout << H3[i] << ' ', std::cout << std::endl;

// Compute sum of elements in D2; sum = 55
int sum = thrust::reduce(D2.begin(), D2.end());
std::cout << 'Sum of D2: ' + std::to_string(sum) + ', std::cout << std::endl;

return 0;
```
Five Operations You Can Do with a Lot of Data in Parallel

- Generate/Create
  - Automatically fill with programmatically defined data

- Transform
  - Apply some “operation” to each element of the data
  - Also called “Map” in many contexts

- Compact
  - Take only the elements in which you are interested
  - Also called “Filter” in many contexts

- Expand
  - The opposite of Compact
  - Create a larger data set from a smaller data set

- Aggregate
  - Calculate a “summary” of your data (e.g., sum or average)
  - Also called “Reduce” or “Fold”
  - “Scan” also provides all intermediate values
Simple Examples with Thrust Pseudocode

- **Generate**
  
  ```cpp
  thrust::sequence(0,4)  0  1  2  3  4
  ```

- **Transform**
  
  ```cpp
  input                  4  5  2  1  3
  thrust::transform(+1)  5  6  3  2  4
  ```

- **Compact**
  
  ```cpp
  input                  4  5  2  1  3
  thrust::copy_if(even)  4  2
  ```

- **Expand**
  
  ```cpp
  input                  4  5  2  1  3
  thrust::for_each(x,2x) 4  8  5 10  2  4  1  2  3  6
  ```

- **Aggregate**
  
  ```cpp
  input                  4  5  2  1  3
  thrust::reduce(+)      15
  ```
Generate Data in Parallel

- Many copies of a certain constant value
  
  ```cpp
  // Ten elements with initial value of integer 1
  thrust::device_vector<int> x(10, 1);
  ```

- A sequence of increasing or decreasing values
  
  ```cpp
  // Allocate space for ten integers, uninitialized
  thrust::device_vector<int> y(10);
  // Fill the space with integers
  thrust::sequence(y.begin(), y.end(), 5, 2);
  ```

- Random Values
  
  - Multiple copies of a random number generator
  - Give each one a different seed
Transform: Vector Addition

- Apply a binary operator “plus” to each element in x and y

```cpp
- thrust::transform(x.begin(), x.end(), // begin and end of the
  y.begin(), // begin of the second
  result.begin(), // begin of the result
  thrust::plus<int>()); // predefined integer
```

-  
  x: 1 1 1 1 1 1 1 1 1 1 
  +
  y: 5 7 9 11 13 15 17 19 21 23 
  =
  result: 6 8 10 12 14 16 18 20 22 24
Transform: Uniform Sampling of a Mathematical Function

- Q: How are we going to generate something more complicated?
  A: Start from some sequence and apply some transformation

- Sampling the function \( f(x) = x^2 \) in the interval of \([0, 1]\)

  ```
  // Generate a sequence of \( x_i \) in \([0,1]\) with \( dx=0.1 \)
  // in between each of them
  float dx = 1.0f/10.0f;
  thrust::sequence(x.begin(), x.end(), 0.0f, dx);

  // Apply the square operation to each of the \( x_i \)
  // to transform into \( f(x_i) = y_i = x_i^2 \)
  thrust::transform(x.begin(), x.end(),
                    y.begin(),
                    square());
  ```

  \[\begin{align*}
  x: & \quad 0.0 \quad 0.1 \quad 0.2 \quad 0.3 \quad 0.4 \quad 0.5 \quad 0.6 \quad 0.7 \quad 0.8 \quad 0.9 \quad 1.0 \\
  y: & \quad 0.0 \quad 0.01 \quad 0.04 \quad 0.09 \quad 0.16 \quad 0.25 \quad 0.36 \quad 0.49 \quad 0.64 \quad 0.81 \quad 1.0
  \end{align*}\]
Reduce: Simple Numerical Integration

- Apply what we learned to estimate the area under a curve
- Create a constant vector of widths
- Create a vector of heights from the function values
- Apply multiply operation on each element of width and height
- Sum all the computed areas to get the total area
- In calculus, this is a method of estimating the integral

\[
\int_0^1 f(x)\,dx \approx \sum_{i=1}^n f(x_i)\Delta x
\]
Simple Numerical Integration: Example

```cpp
thrust::device_vector<int> width(11, 0.1);
width       = 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1

thrust::sequence(x.begin(), x.end(), 0.0f, 0.1f);
x           = 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

thrust::transform(x.begin(), x.end(), height.begin(), square());
height      = 0.0 0.01 0.04 0.09 0.16 0.25 0.36 0.49 0.64 0.81 1.0

thrust::transform(width.begin(), width.end(), height.begin(), area.begin(),
thrust::multiplies<float>())
area        = 0.0 0.001 0.004 0.009 0.016 0.025 0.036 0.049 0.064 0.081 0.1

total_area = thrust::reduce(area.begin(), area.end());
total_area = 0.385

thrust::inclusive_scan(area.begin(), area.end(), accum_areas.begin());
accum_areas = 0.0 0.001 0.005 0.014 0.030 0.055 0.091 0.140 0.204 0.285 0.385
```
Scan: Simple Numerical Integration

- What happens if we are interested in the integral $F(t) = F(0) + \int_0^t f(x)dx$ on the interval [0, 1] instead of just a number?

- Calculate a running sum by using scan

```cpp
thrust::inclusive_scan(y_dx.begin(), y_dx.end(),
                        F.begin());
```

- $f(x_i)dx = 0.0 0.001 0.004 0.009 0.016 0.025 0.036 0.049 0.064 0.081 \ldots 0.1$

- $F(t) = 0.0 0.001 0.005 0.014 0.030 0.055 0.091 0.140 0.204 0.285 0.385$

- The last element of the scan (0.385) is the same as the output of reduce

- In mathematical terms,

$$\int_0^1 f(x)dx = F(1) - F(0)$$
Scan: Calculating the Fibonacci Sequence by Matrix Multiplication

- The reduce and scan operators can also work with a user defined type.
- The Fibonacci Sequence is defined as
  \[ F_{n+1} = F_n + F_{n-1} \quad \text{with} \quad F_0 = 0, \ F_1 = 1 \]
- By “unrolling” the recurrence we have
  \[
  \begin{bmatrix}
  F_{n+1} \\
  F_n
  \end{bmatrix}
  =
  \begin{bmatrix}
  1 & 1 \\
  1 & 0
  \end{bmatrix}
  \begin{bmatrix}
  F_n \\
  F_{n-1}
  \end{bmatrix}
  \]
- Thus we can compute \( F_n \) by matrix multiplication
  \[
  \begin{bmatrix}
  1 & 1 \\
  1 & 0
  \end{bmatrix}
  \begin{bmatrix}
  1 & 1 & 1 & 1 & 1 & 1 \\
  1 & 0 & 1 & 0 & 1 & 0
  \end{bmatrix}
  =
  \begin{bmatrix}
  1 & 1 & 2 & 3 & 5 & 8 \\
  1 & 0 & 1 & 1 & 2 & 3
  \end{bmatrix}
  \]
# Fibonacci Sequence using a Matrix Scan

```cpp
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <thrust/scan.h>
#include <thrust/transform.h>
#include <thrust/iterator/constant_iterator.h>

// A simple data structure for 2x2 matrix

template <typename T>
struct mat22 {
    T m00, m10, m01, m11;
    __host__ __device__ mat22() {}
    __host__ __device__ mat22(T a, T b, T c, T d) : m00(a), m10(b), m01(c), m11(d) {}
};

// Overload the multiplication operator for 2x2 matrices

template <typename T>
__host__ __device__ operator*(const mat22<T> &lhs, const mat22<T> &rhs) {
    return mat22<T>{
        lhs.m00*rhs.m00+lhs.m01*rhs.m10,
        lhs.m00*rhs.m01+lhs.m01*rhs.m11,
        lhs.m10*rhs.m00+lhs.m11*rhs.m10,
        lhs.m10*rhs.m01+lhs.m11*rhs.m11
    };
}

// Wrap the overloaded multiplication operator into a binary functor

template <typename T>
struct matmul : public thrust::binary_function<mat22<T>, mat22<T>, mat22<T> > {
    __host__ __device__ mat22<T> operator()(const mat22<T> &lhs, const mat22<T> &rhs) {
        return lhs*rhs;
    }
};

// Extract the (0, 0) element from the 2x2 matrix

template <typename T>
struct zeroth_elam : public thrust::unary_function<mat22<T>, T> {
    __host__ __device__
    T operator()(const mat22<T> &mat) {
        return mat.m00;
    }
};

int main(void) {
    const int N = 20;
    mat22<unsigned long> A(1, 1, 1, 0);
    thrust::constant_iterator<mat22<unsigned long> > begin(A);
    thrust::device_vector<mat22<unsigned long> > vec(N);

    // we multiply a bunch of matrix A together.
    thrust::inclusive_scan(begin, begin + N, vec.begin(),
        matmul<unsigned long>(),
        thrust::transform(vec.begin(), vec.end(),
            fib.begin(),
            zeroth_elam<unsigned long>(),
            thrust::device_vector<unsigned int> fib(N),
            thrust::transform(vec.begin(), vec.end(),
                fib.begin(),
                zeroth_elam<unsigned long>(),
                std::ostream_iterator<unsigned long>(std::cout, " "))));
    std::cout << std::endl;
}
```

LA-UR-14-20028
Distributed Scan Algorithm

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>13</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>15</td>
<td>18</td>
</tr>
</tbody>
</table>

thrust::inclusive_scan

MPI_Gather

MPI_Scatter

thrust::transform

Alternative: MPI_Exscan

LA-UR-14-20028
Tree Scan Algorithm

Up-sweep (reduce)
for d = 0 to \( \log_2 n - 1 \) do
    for k from 0 to n-1 by 2^{d+1} pardo
        \( x[k+2^{d+1}-1] = x[k+2^d-1] + x[k+2^{d+1}-1] \)

Down-sweep
for d = \( \log_2 n - 1 \) down to 0 do
    for k from 0 to n-1 by 2^{d+1} pardo
        t = x[k+2^d-1]
        \( x[k+2^{d+1}-1] = t + x[k+2^{d+1}-1] \)
Compaction: Finding Prime Numbers Using the Sieve of Eratosthenes

- Start with a vector containing the sequence of integers from 2 to N
- The first element in this vector is prime
- Use compaction to copy only elements of the vector not divisible by this prime into an updated vector (Thrust copy_if operator)
- The second element in this vector is prime
- Repeat the two steps above until you reach the end of the vector
Boid Simulation

- Simulate flocking behavior of a group of “boids”

- At each time step, velocities are adjusted based on three parameters, each dependent only upon observing other nearby boids:
  
  - Cohesion: Each boid wants to move towards the centroid of other boids in its vicinity, to join the flock
  
  - Separation: Each boid wants to move away from other boids that are too close to it, to avoid collisions
  
  - Alignment: Each boid wants to adjust its velocity (direction and magnitude) to match that of other boids in its vicinity, to move in sync with the flock

- Positions are then updated for the next time step based on the new velocities

- Thrust transform functions are used in order to parallelize the computations for all the boids

- Functors are used to compute the cohesion, separation, and alignment parameters for a boid, and to update its velocity and position

Boid Simulation Video
class flock_sim
{
    struct cohesion {...}
    struct separation {...}
    struct alignment {...}
    struct updateVelocity {...}
    struct updatePosition {...}
    struct bounce {...}
    thrust::device_vector<float3> m_positions, m_velocities;
    thrust::device_vector<float3> m_cohesion, m_separation, m_alignment;
    float m_cohesionWeight, m_separationWeight, m_alignmentWeight;
    ...
    flock_sim(...){...}
    void operator()(){...}
};
Main simulation loop

```cpp
void operator()()
{
    // Compute the cohesion term for the velocity update
    thrust::transform(m_positions.begin(), m_positions.end(), m_cohesion.begin(),
                      cohesion(m_n, m_cohesionThresholdSq, thrust::raw_pointer_cast(&m_positions.begin())));

    // Compute the separation term for the velocity update
    thrust::transform(m_positions.begin(), m_positions.end(), m_separation.begin(),
                      separation(m_n, m_separationThresholdSq, thrust::raw_pointer_cast(&m_positions.begin())));

    // Compute the alignment term for the velocity update
    thrust::transform(thrust::make_zip_iterator(thrust::make_tuple(m_positions.begin(), m_velocities.begin()))),
                      thrust::make_zip_iterator(thrust::make_tuple(m_positions.end(), m_velocities.end()))),
                      m_alignment.begin()),
                      alignment(m_n, m_alignmentThresholdSq, thrust::raw_pointer_cast(&m_positions.begin()),
                               thrust::raw_pointer_cast(&m_velocities.begin())));

    // Update the velocity based on the computed cohesion, separation, and alignment adjustments
    thrust::transform(thrust::make_counting_iterator(0), thrust::make_counting_iterator(0)+m_n, m_velocities.begin(),
                      updateVelocity(m_cohesionWeight, m_separationWeight, m_alignmentWeight, m_velocityScale,
                                      thrust::raw_pointer_cast(&m_cohesion.begin()),
                                      thrust::raw_pointer_cast(&m_separation.begin()),
                                      thrust::raw_pointer_cast(&m_alignment.begin()),
                                      thrust::raw_pointer_cast(&m_velocities.begin())));

    // Update the bold positions based on the new velocities for this time step
    thrust::transform(thrust::make_zip_iterator(thrust::make_tuple(m_positions.begin(), m_velocities.begin()))),
                      thrust::make_zip_iterator(thrust::make_tuple(m_positions.end(), m_velocities.end()))),
                      m_positions.begin(), updatePosition(m_dt, m_minSpeed, m_maxSpeed));

    // Clamp any boids that have moved outside the simulation boundaries, and reverse their velocities so they bounce back inside
    thrust::for_each(thrust::make_counting_iterator(0), thrust::make_counting_iterator(0)+m_n,
                     bounce(m_boundaryMin, m_boundaryMax, thrust::raw_pointer_cast(&m_positions.begin()),
                            thrust::raw_pointer_cast(&m_velocities.begin())));
}
```
struct cohesion : public thrust::unary_function<float3, float3>
{
    int n;
    float thresholdSq;
    float3* positions;

    __host__ __device__
    cohesion(int n, float thresholdSq, float3* positions) : n(n), thresholdSq(thresholdSq), positions(positions) {
    }

    __host__ __device__
    float3 operator()(float3 a_position) const
    {
        // Compute centroid of all neighbors by searching through all other boids
        float3 centroid = make_float3(0.0f, 0.0f, 0.0f);
        int neighbors = 0;
        for (unsigned int i=0; i<n; i++)
        {
            float3 diff = a_position - positions[i];
            if (dot(diff, diff) < thresholdSq)
            {
                centroid = centroid + positions[i];
                neighbors++;
            }
        }
        if (neighbors == 0) return make_float3(0.0f, 0.0f, 0.0f);
        centroid.x /= neighbors; centroid.y /= neighbors; centroid.z /= neighbors;

        // Add a term to the velocity pointed towards the centroid of the neighbors
        return (centroid - a_position);
    }
};
Separation Term

```cpp
struct separation : public thrust::unary_function<float3, float3>
{
    int n;
    float thresholdSq;
    float3* positions;

    __host__ __device__
    separation(int n, float thresholdSq, float3* positions) : n(n), thresholdSq(thresholdSq), positions(positions) { }

    __host__ __device__
    float3 operator()(float3 a_position) const
    {
        // Add a term to the velocity pointed away from each neighbor that is too close
        float3 repel = make_float3(0.0f, 0.0f, 0.0f);
        for (unsigned int i=0; i<n; i++)
        {
            float3 diff = a_position-positions[i];
            if (dot(diff, diff) < thresholdSq && (dot(diff, diff) > NEAR_ZERO))
                repel = repel + normalize(diff);
        }
        if (dot(repel, repel) < NEAR_ZERO) return make_float3(0.0f, 0.0f, 0.0f);
        return normalize(repel);
    }
};
```
struct alignment : public thrust::unary_function<thrust::tuple<float3, float3>, float3>
{
    int n;
    float thresholdSq;
    float3 *positions, *velocities;

    __host__ __device__
    alignment(int n, float thresholdSq, float3* positions, float3* velocities):
        n(n), thresholdSq(thresholdSq), positions(positions), velocities(velocities) { };

    __host__ __device__
    float3 operator()(thrust::tuple<float3, float3> a_posAndVel) const
    {
        // Extract the position and the velocity from the tuple
        float3 a_position = thrust::get<0>(a_posAndVel);
        float3 a_velocity = thrust::get<1>(a_posAndVel);

        // Compute the average velocity for all neighbors by searching through all other boids
        float3 avgVelocity = make_float3(0.0f, 0.0f, 0.0f);
        int neighbors = 0;
        for (unsigned int i=0; i<n; i++)
        {
            float3 diff = a_position - positions[i];
            if (dot(diff, diff) < thresholdSq)
            {
                avgVelocity = avgVelocity + velocities[i];
                neighbors++;
            }
        }
        if (neighbors == 0) return make_float3(0.0f, 0.0f, 0.0f);
        avgVelocity.x /= neighbors; avgVelocity.y /= neighbors; avgVelocity.z /= neighbors;

        // Add a term to the velocity to make it closer to the average velocity of the neighbors
        return (avgVelocity + a_velocity);
    }
};
Updating Velocities

\[
v_{t+1} = v_t + w_v \left( w_c c + w_s s + w_a a \right)
\]

```cpp
t struct updateVelocity : public thrust::unary_function<int, float3>
{
    float cohesionWeight, separationWeight, alignmentWeight, velocityAdjustmentScale;

    _host_ _device_
    updateVelocity(float cohesionWeight, float separationWeight, float alignmentWeight, float velocityAdjustmentScale,
            float3* cohesion, float3* separation, float3* alignment, float3* velocities):
        cohesionWeight(cohesionWeight), separationWeight(separationWeight), alignmentWeight(alignmentWeight),
        velocityAdjustmentScale(velocityAdjustmentScale), cohesion(cohesion), separation(separation),
        alignment(alignment), velocities(velocities) { }

    _host_ _device_
    float3 operator()(int i) const
    {
        // Adjust the velocity based on the cohesion, separation, and alignment terms and their weights
        float3 newVelocity = (velocities[i] + velocityAdjustmentScale*(cohesionWeight*cohesion[i] +
            separationWeight*separation[i] + alignmentWeight*alignment[i]));
        return newVelocity;
    }
};
```
Updating Positions

```c++
struct updatePosition : public thrust::unary_function<thrust::tuple<float3, float3>, float3>
{
    float dt, minSpeed, maxSpeed;

    __host__ __device__
    updatePosition(float velocityScale, float minSpeed, float maxSpeed) : dt(dt),
    minSpeed(minSpeed), maxSpeed(maxSpeed) {};

    __host__ __device__
    float3 operator()(thrust::tuple<float3, float3> a_posAndVel) const
    {
        // Extract the position and the velocity from the tuple, and clamp the velocity between minimum and maximum values
        float3 a_position = thrust::get<0>(a_posAndVel);
        float3 a_velocity = thrust::get<1>(a_posAndVel);
        if (dot(a_velocity, a_velocity) > maxSpeed*maxSpeed) a_velocity = maxSpeed*normalize(a_velocity);
        if (dot(a_velocity, a_velocity) < minSpeed*minSpeed) a_velocity = minSpeed*normalize(a_velocity);

        // Update the position based on the velocity computed by this timestep
        return (a_position + a_velocity*dt);
    }
};
```

\[ x_{t+1} = x_t + v_t \Delta t \]
Bouncing off boundaries

```cpp
struct bounce : public thrust::unary_function<int, void>
{
    float3 clampMin, clampMax;
    float3 *positions, *velocities;

    __host__ __device__
    bounce(float3 clampMin, float3 clampMax, float3* positions, float3* velocities):
    clampMin(clampMin), clampMax(clampMax), positions(positions), velocities(velocities) {};

    __host__ __device__
    void operator()(int i) const
    {
        // If the boid has moved outside the simulation boundaries, clamp it inside and reverse its velocity
        float3 result = positions[i];
        bool bounce = false;
        if (result.x < clampMin.x) { bounce = true; result.x = clampMin.x; }
        if (result.x > clampMax.x) { bounce = true; result.x = clampMax.x; }
        if (result.y < clampMin.y) { bounce = true; result.y = clampMin.y; }
        if (result.y > clampMax.y) { bounce = true; result.y = clampMax.y; }
        if (result.z < clampMin.z) { bounce = true; result.z = clampMin.z; }
        if (result.z > clampMax.z) { bounce = true; result.z = clampMax.z; }
        positions[i] = result;
        if (bounce) velocities[i] = -1.0f*velocities[i];
    }
};
```
Interop

- Without interop, separate memory is used on the GPU for computation results and for rendering, and data transfer goes through the CPU.
- With interop, a shared region of memory on the GPU is used both for computation and for rendering, eliminating the slow GPU-CPU data transfers.
Conclusion

- The example codes we showed are independent of the location of data and execution
- It can be executed serially on CPU or parallel backends
- Debug on CPU during development; use parallel execution in “production”
- Extend to other languages and libraries
  - STL in C++
  - Copperhead in Python
  - SQL/LINQ for databases
High-Level Data-Parallelism Future-Proofs Your Code

- The high-level parallel algorithms you write today will still work with new hardware in the future.
- In fact, they will only get faster!
- The skills you learn in developing high-level parallel algorithms will still be applicable in the future even as computing technology improves.
Obtaining Thrust and Data-Parallel Examples

- Obtain Thrust:
  - Part of CUDA: https://developer.nvidia.com/cuda-downloads
  - Or, just get Thrust: http://thrust.github.com

- Obtain the examples from this presentation from github: https://github.com/losalamos/PISTON/tree/master/tutorial/NMSCC13

- If you have questions, contact Chris (csewell@lanl.gov)
Instructions for Example Code

- This project uses CMake (available at http://www.cmake.org/cmake/resources/software.html). Create a build directory, and run ccmake “path-to-source-directory”.

- Run with or without CUDA
  
  - To run the CUDA backend, you need an NVIDIA GPU (that is CUDA-capable) and an installation of NVIDIA's CUDA toolkit (available at https://developer.nvidia.com/cuda-downloads), and the location of this installation should be provided in the CMake configuration.

  - Alternatively, you can run just the OpenMP backend by downloading Thrust 1.7 rather than CUDA (available at thrust.github.com). In the CMake configuration, set USE_CUDA to OFF, and provide the location of Thrust in the CMake configuration for the THRUST_DIR CMake variable. If using a Mac, and your default compiler uses Clang, you need to use set the old gcc (/usr/bin/g++) instead of letting CMake to figure which compiler to use.

- Once you have set all of the CMake variables, select the configure and then generate options. You can then quit CMake and run make to build the examples.
RESEARCH TALK OVERVIEW OF PISTON AND PINION
SDAV VTK-m Frameworks

- Objective: Enhance existing multi/many-core technologies in anticipation of in situ analysis use cases with LCF codes

- Benefit to scientists: These frameworks will make it easier for domain scientists’ simulation codes to take advantage of the parallelism available on a wide range of current and next-generation hardware architectures, especially with regards to visualization and analysis tasks

- Projects
  - EAVL, Oak Ridge National Laboratory
  - DAX, Sandia National Laboratory
  - DIY, Argonne National Laboratory
  - PISTON, Los Alamos National Laboratory

- Work on integrating these projects with VTK is on-going, in collaboration with Kitware
PISTON: A Portable Data-Parallel Visualization and Analysis Framework

- Goal: Portability and performance for visualization and analysis operators on current and next-generation supercomputers
- Main idea: Write operators using only data-parallel primitives (scan, reduce, etc.)
- Requires architecture-specific optimizations for only for the small set of primitives
- PISTON is built on top of NVIDIA's Thrust Library
Motivation and Background

- Current production visualization software does not take full advantage of acceleration hardware and/or multi-core architecture.

- Research on accelerating visualization operations are mostly hardware-specific; few were integrated in visualization software.

- Standards such as OpenCL may allow program to run cross-platform, but usually still requires many architecture specific optimizations to run well.

- Data parallelism: independent processors performs the same task on different pieces of data (see Blelloch, “Vector Models for Data Parallel Computing”).

- Due to the massive data sizes we expect to be simulating we expect data parallelism to be a good way to exploit parallelism on current and next generation architectures.

- Thrust is a NVidia C++ template library for CUDA. It can also target other backends such as OpenMP, and allows you to program using an interface similar the C++ Standard Template Library (STL).
Videos of PISTON in Action
Brief Introduction to Data-Parallel Programming and Thrust

What algorithms does Thrust provide?

- Sorts
- Transforms
- Reductions
- Scans
- Binary searches
- Stream compactions
- Scatters / gathers

**Challenge:** Write operators in terms of these primitives only

**Reward:** Efficient, portable code

---

```
input          4 5 2 1 3
transform(+1)  5 6 3 2 4
inclusive_scan(+)  4 9 11 12 15
exclusive_scan(+)   0 4 9 11 12
exclusive_scan(max) 0 4 5 5 5
transform_inscan(*2,+)*2,+) 8 18 22 24 30
for_each(-1) 3 4 1 0 2
sort          1 2 3 4 5
copy_if(n % 2 == 1) 5 1 3
reduce(+)             15

input1         0 0 2 4 8
input2         3 4 1 0 2
upper_bound    3 4 2 2 3
permutation_iterator  4 8 0 0 2
```
Isosurface with Marching Cube – the Naive Way

- Classify all cells by `transform`
- Use `copy_if` to compact valid cells.
- For each valid cell, generate same number of geometries with flags.
- Use `copy_if` to do stream compaction on vertices.
- This approach is too slow, more than 50% of time was spent moving huge amount of data in global memory.
- Can we avoid calling `copy_if` and eliminate global memory movement?
Isosurface with Marching Cube – Optimization

- Inspired by HistoPyramid
- The filter is essentially a mapping from input cell id to output vertex id
- Is there a “reverse” mapping?
- If there is a reverse mapping, the filter can be very “lazy”
- Given an output vertex id, we only apply operations on the cell that would generate the vertex
- Actually for a range of output vertex ids
Isosurface with Marching Cubes Algorithm

1. input
   `transform(classify_cell)`
2. caseNums
   4 0 12 0 6 0 5
3. numVertices
   `transform_inclusive_scan(is_valid_cell)`
   2 0 2 0 2 0 4
4. validCellEnum
   1 1 2 2 3 3 4
5. CountingIterator
   `upper_bound`
6. validCellIndices
   0 1 2 3
7. numVerticesCompacted
   `exclusive_scan`
   2 2 2 4
8. numVerticesEnum
   `for_each(isosurface_functor)`
9. outputVertices
   0 1 2 3 4 5 6 8 9

# of valid cells = 4
Total # of vertices = 10
Variations on Isosurface: Cut Surfaces and Threshold

- Cut surface
  - Two scalar fields, one for generating geometry (cut surface) the other for scalar interpolation
  - Less than 10 LOC change, negligible performance impact to isosurface
  - One 1D interpolation per triangle vertex

- Threshold
  - Classify cells, this time based on whether value at each vertex falls within threshold range, then stream compact valid cells and generate geometry for valid cells
  - Additional pass of cell classification and stream compaction to remove interior cells
Additional Operators

Belloch’s “Vector Models for Data-Parallel Computing”

Data Structures
- Graphs: Neighbor reducing, distributing excess across edges
- Trees: Leaffix and rootfix operations, tree manipulations
- Multidimensional arrays

Computational Geometry
- Generalized binary search
- k-D tree
- Closest pair
- Quickhull
- Merge Hull

Graph Algorithms
- Minimum spanning tree
- Maximum flow
- Maximal independent set

Numerical Algorithms
- Matrix-vector multiplication
- Linear-systems solver
- Simplex
- Outer product
- Sparse-matrix multiplication

Our implementations
- Glyphs
- KD-Tree Construction
- Halo finder for cosmology simulations
- “Boid” simulation (flocking birds)

http://www.cs.cmu.edu/~blelloch/papers/Ble90.pdf
PISTON Performance

3D Isosurface Generation: CUDA Compute Rates
- NVIDIA Native CUDA Demo (Quadro 448 cores)
- PISTON CUDA Backend (Quadro 448 cores)

3D Isosurface Generation: CPU Compute Rates
- PISTON OMP Backend (Opteron 48 cores)
- Parallel VTK (Opteron 48 cores)
- VTK (Opteron 1 core)
Integration with VTK and ParaView

- Filters that use PISTON data types and algorithms integrated into VTK and ParaView
- Utility filters interconvert between standard VTK data format and PISTON data format (thrust device vectors)
- Supports interop for on-card rendering
Extending PISTON’s Portability: Architectures (1/3)

- Prototype OpenCL backend
  - Successfully implemented isosurface and cut plane operators in OpenCL with code almost identical to that used for the Thrust-based CUDA and OpenMP backends
  - With interop on AMD FirePro V7800, we can run at about 6 fps for $256^3$ data set (2 fps without interop)

- Renderer
  - Allows generation of images on systems without OpenGL
  - Rasterizing and ray-casting versions (using KD-Tree)
Extending PISTON’s Portability: Architectures (2/3)

- Inter-node (distributed memory) parallelism
  - VTK Integration handles domain decomposition / image compositing
  - Distributed implementations of Thrust primitives using MPI
    - User can treat data as single vectors even though values are distributed across nodes
    - Regular Thrust primitives are called for on-node work, so it takes advantage of parallelism both on nodes and across nodes
    - Implemented isosurface and KD-tree construction algorithms using distributed PISTON

Isosurface of 3600x2400x42 ocean temperature data computed on 4 GPUs
Extending PISTON’s Portability: Architectures (3/3)
Extending PISTON’s Portability: Data Types

- Curvilinear coordinates
  - Multiple layers of coordinate transformations
  - Due to kernel fusion, very little performance impact

- Unstructured / AMR data
  - Tetrahedralize uniform grid or unstructured grid (e.g., AMR mesh)
  - Generate isosurface geometry based on look-up table for tetrahedral cells
  - Next step: Develop PISTON operator to tetrahedralize grids, and/or to compute isosurface directly on AMR grid
PISTON In-Situ

- VPIC (Vector Particle in Cell) Kinetic Plasma Simulation Code
  - Implemented an in-situ adapter based on Paraview CoProcessing Library (Catalyst)
  - PISTON contour pipeline using ParaView’s PISTON integration
- CoGL
  - Stand-alone meso-scale simulation code developed as part of the Exascale Co-Design Center for Materials in Extreme Environments
  - Studies pattern formation in ferroelastic materials using the Ginzburg–Landau approach
  - Models cubic-to-tetragonal transitions under dynamic strain loading
  - Simulation code and in-situ viz implemented using PISTON

Output of PISTON contour filter on Hhydro charge density at one timestep of VPIC simulation

PISTON in-situ visualization of CoGL Ginzburg-Landau simulation
PISTON’s Companion Project: PINION

- A portable, data-parallel software framework for physics simulations
  - Data structures that allow scientists to program in a way that maps easily to the problem domain rather than dealing directly with 1D host/device vectors
  - Operators that provide data-parallel implementations of analysis and computational functions often used in physics simulations
  - Backends that optimize implementations of data parallel primitives for one or two emerging supercomputer hardware architectures

\[
\frac{\partial f}{\partial t} + \nabla(uf) = 0 \quad \text{(Advection)} \\
\rho \left( \frac{\partial v}{\partial t} + v \cdot \nabla v \right) = -\nabla p + \nabla \cdot T + \mathbf{f} \quad \text{(Navier-Stokes)}
\]
ADDITIONAL TUTORIAL EXAMPLES
Tutorials

- Be a Thrust user
  - Thrust QuickStartGuide examples

- Be a PISTON user
  - tutorial1{OMP/GPU}: create a tangle field and apply the PISTON isosurface operator
  - demo{OMP/GPU}: load a VTK structured grid from a file and apply the PISTON isosurface, cut plane, or threshold operator

- Be a PISTON developer
  - tutorial2{OMP/GPU}: write a simple simulation (boid flocking) and a simple visualization operator (glyphs) using Thrust primitives, and chain them together (optionally using interop with CUDA)

- Be a data-parallel algorithm designer using PISTON
  - tutorial3{OMP/GPU}: use data parallel primitives to design a KD-tree algorithm
Tutorial 1: Use PISTON isosurface operator (1/2)

```c
void GLWindow::initializeGL()
{
  // Initialize camera and UI variables
  qrot.set(0,0,0,1);
  frameCount = 0;
  gridSize = 64;
  cameraFOV = 60.0f;
  minIsovalue = 31.0f;
  maxIsovalue = 500.0f;
  wireframe = false;

  // Set up basic OpenGL state and lighting
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glEnable(GL_DEPTH_TEST);
  glShadeModel(GL_SMOOTH);
  float white[] = { 0.8, 0.8, 0.8, 1.0 };
  float black[] = { 0.0, 0.0, 0.0, 1.0 };
  float lightPos[] = { 0.0, 0.0, gridSize*1.5, 1.0 };
  glEnable(GL_LIGHT0);
  glLightfv(GL_LIGHT0, GL_AMBIENT, white);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, white);
  glLightfv(GL_LIGHT0, GL_SPECULAR, black);
  glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
  glEnable(GL_LIGHTMODEL_TWO_SIDE, 1);
  glEnable(GL_LIGHTING);
  glEnable(GL_NORMALIZE);
  glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
  glEnable(GL_COLOR_MATERIAL);

  // Initialize CUDA if it is being used
  #ifdef USE_INTEROP
    glewInit();
    cudaGLSetDevice(0);
  #endif

  // Create the tangle field and marching cube operator instances
  tangle = new tangle_field<int, float, SPACE>(gridSize, gridSize, gridSize);
  isosurface = new marching_cube<tangle_field<int, float, SPACE>>();
  tangle_field<int, float, SPACE> (*tangle, *tangle, 0.2f);

  // Compute the isosurface of the tangle field
  (*isosurface());

  // If using interop, generate the vertex buffer objects to be shared between CUDA and OpenGL
  #ifdef USE_INTEROP
    int numPoints = thrust::distance(isosurface->vertices_begin(), isosurface->vertices_end());
    glBindBuffer(GL_ARRAY_BUFFER, vboBuffers[0]);
    for (int i = 0; i < 4; ++i)
    {
      unsigned int bufferSize = (i == 2) ? numPoints*sizeof(float3) : numPoints*sizeof(float4);
      glBindBuffer(GL_ARRAY_BUFFER, vboBuffers[i]);
      glBufferData(GL_ARRAY_BUFFER, bufferSize, 0, GL_DYNAMIC_DRAW);
    }
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    for (int i = 0; i < 3; ++i)
    {
      cudaMemcpyGlRegisterBuffer<vboResources[i]>, vboBuffers[i], cudaGraphicsMapFlags::discard);
      cudaMemcpyGlRegisterBuffer<vboResources[i] = vboResources[i] >= vboResources[i] + sizeof(float4));
      isosurface->vboResources[i] = vboResources[i];
    }
    isosurface->minIso = minIsovalue; isosurface->maxIso = maxIsovalue;
    isosurface->useInterop = true; isosurface->vboSize = numPoints;
  #endif

  // Enable OpenGL state for vertex, normal, and color arrays
  glEnableClientState(GL_VERTEX_ARRAY);
  glEnableClientState(GL_NORMAL_ARRAY);
  glEnableClientState(GL_COLOR_ARRAY);
```
Tutorial 1: Use PISTON isosurface operator (2/2)

```c
void GLWindow::paintGL()
{
    // Stop the QT callback timer
    timer->stop();

    // Start timing this interval
    if (frameCount == 0) gettimeofday(&begin, 0);

    // Set up the OpenGL state
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    if (wireframe) glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    else glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

    // Set up the projection and modelview matrices for the view
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(cameraFOV, 1.0f, 1.0f, gridScale*4.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0f, 0.0f, gridScale*1.5f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);

    // Set up the current rotation and translation
    qrot.setRotMat(rotationMatrix);
    gmlRotMat = qrot.getRotMat();
    gmlTrans((gridScale-1)/2, -(gridScale-1)/2, -(gridScale-1)/2);

    // Compute the isosurface of the tangle field
    (*isosurface);

    // If using interop, render the vertex buffer objects; otherwise, render the arrays
    #ifdef USE_INTEROP
        glBindBuffer(GL_ARRAY_BUFFER, vboBuffers[0]);
        glVertexPointer(4, GL_FLOAT, 0, 0);
        glBindBuffer(GL_ARRAY_BUFFER, vboBuffers[2]);
        glNormalPointer(GL_FLOAT, 0, 0);
        glBindBuffer(GL_ARRAY_BUFFER, vboBuffers[1]);
        glColorPointer(4, GL_FLOAT, 0, 0);
        glDrawArrays(GL_TRIANGLES, 0, isosurface->vboSize);
    #else
        vertices.assign(isosurface->vertices_begin(), isosurface->vertices_end());
        normals.assign(isosurface->normals_begin(), isosurface->normals_end());
        colors.assign(thrust::make_transform_iterator(isosurface->normals_begin(),
                                                       color_map<float>).(31.0f, 500.0f)),
        thrust::make_transform_iterator(isosurface->normals_end(),
                                                       color_map<float>).(31.0f, 500.0f));
        glColorPointer(4, GL_FLOAT, 0, &colors[0]);
        glNormalPointer(0, 0, &normals[0]);
        glVertexPointer(4, GL_FLOAT, 0, &vertices[0]);
        glDrawArrays(GL_TRIANGLES, 0, vertices.size());
    #endif

    // Pop this OpenGL view matrix
    glPopMatrix();

    // Periodically output the framerate
    gettimeofday(&end, 0);
    timersub(&end, &begin, &diff);
    frameCount++;
    float seconds = diff.tv_sec + 1.0e-6*diff.tv_usec;
    if (seconds > 0.5f)
    {
        char title[256];
        sprintf(title, "Marching Cube, fps: %2.2f", float(frameCount)/seconds);
        std::cout << title << std::endl;
        seconds = 0.0f;
        frameCount = 0;
    }

    // Restart the QT callback timer
    timer->start(1);
}
```
Tutorial 2: Boid Simulation

- Already covered in first section
Tutorial 3: KD-Tree
## KD Tree: Overview

<table>
<thead>
<tr>
<th>Level 1</th>
<th>Point Ids</th>
<th>X Ranks</th>
<th>Y Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>computeGlobalRanks</td>
<td>0 1 2 3 4 5 6 7</td>
<td>1 6 0 2 7 3 4 5</td>
<td>0 5 2 3 7 6 4 1</td>
</tr>
<tr>
<td>computeFlags</td>
<td>F T F F T F T T</td>
<td>F T F F T F T T</td>
<td>F T F F T F T T</td>
</tr>
<tr>
<td>segmentedSplit</td>
<td>0 2 3 5 1 4 6 7</td>
<td>1 0 2 3 6 7 4 5</td>
<td>0 2 3 6 5 7 4 1</td>
</tr>
<tr>
<td>renumberRanks</td>
<td>1 0 2 3 2 3 0 1</td>
<td>0 1 2 3 2 3 1 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level 2</th>
<th>Point Ids</th>
<th>X Ranks</th>
<th>Y Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>computeFlags</td>
<td>F F T T T T F F</td>
<td>F F T T T T F F</td>
<td>F F T T T T F F</td>
</tr>
<tr>
<td>segmentedSplit</td>
<td>0 2 3 5 6 7 1 4</td>
<td>1 0 2 3 0 1 2 3</td>
<td>0 1 2 3 1 0 2 3</td>
</tr>
<tr>
<td>renumberRanks</td>
<td>1 0 0 1 0 1 0 1</td>
<td>0 1 0 1 1 0 0 1</td>
<td></td>
</tr>
</tbody>
</table>
## KD Tree: computeGlobalRanks

<table>
<thead>
<tr>
<th></th>
<th>A Input coordinates</th>
<th>2.9 8.9 2.4 6.4 9.3 6.9 7.5 7.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>B CountingIterator(0)</td>
<td>0 1 2 3 4 5 6 7</td>
<td></td>
</tr>
<tr>
<td>C sort_by_key(A,B)</td>
<td>2.4 2.9 6.4 6.9 7.5 7.6 8.9 9.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 0 3 5 6 7 1 4</td>
<td></td>
</tr>
<tr>
<td>D scatter(B,C)</td>
<td>1 6 0 2 7 3 4 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---</td>
<td>--------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>A</td>
<td>Input ranks</td>
<td>1 6 0 2 7 3 4 5</td>
</tr>
<tr>
<td>B</td>
<td>Input segmentIds</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>C</td>
<td>CountingIterator(1)</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>D</td>
<td>Reverse inclusive_scan_by_key(B,C,max)</td>
<td>8 8 8 8 8 8 8 8</td>
</tr>
<tr>
<td></td>
<td>// # elements in segment</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>transform(E[i]=D[i]/2)</td>
<td>4 4 4 4 4 4 4 4</td>
</tr>
<tr>
<td></td>
<td>// # median index</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>transform(F[i]=A[i]&gt;=E[i])</td>
<td>F T F F T F T T</td>
</tr>
</tbody>
</table>

KD Tree: computeFlags
# KD Tree: segmentedSplit

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input pointIds</td>
<td>0 2 3 5 1 4 6 7</td>
</tr>
<tr>
<td>B</td>
<td>Input flags</td>
<td>F F T T T T F F</td>
</tr>
<tr>
<td>C</td>
<td>Input segmentIds</td>
<td>0 0 0 0 1 1 1 1</td>
</tr>
<tr>
<td>D</td>
<td>exclusive_scan_by_key(C,B)</td>
<td>0 0 0 1 0 1 2 2</td>
</tr>
<tr>
<td></td>
<td>// total number of true flags preceding in segment</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>CountingIterator(0)</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>F</td>
<td>inclusive_scan_by_key(C,E,min)</td>
<td>0 0 0 0 4 4 4 4</td>
</tr>
<tr>
<td></td>
<td>// total number of elements in previous segments</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>CountingIterator(1)</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>H</td>
<td>Reverse inclusive_scan_by_key(C,G,max)</td>
<td>4 4 4 4 8 8 8 8</td>
</tr>
<tr>
<td></td>
<td>// index of last element in its segment (+1)</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>inclusive_scan_by_key(C,inverse(B))</td>
<td>1 2 2 2 0 0 1 2</td>
</tr>
<tr>
<td></td>
<td>// total number of false flags so far in segment</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>transform(J[i]=(if(B[i]) F[i]+I[H[i]-1]+D[i] else F[i]+I[i]-1))</td>
<td>0 1 2 3 6 7 4 5</td>
</tr>
<tr>
<td>K</td>
<td>scatter(A,J)</td>
<td>0 2 3 5 6 7 1 4</td>
</tr>
</tbody>
</table>
**KD Tree: renumberRanks**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>0</th>
<th>2</th>
<th>3</th>
<th>6</th>
<th>5</th>
<th>7</th>
<th>4</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input ranks</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>Input flags</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>C</td>
<td>Input segmentIds</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>ConstantIterator(1)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>exclusive_scan_by_key(C,D)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>scatter(E,A)</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>scatter(B,A)</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>H</td>
<td>segmentedSplit(F,G)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>CountingIterator(0)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>J</td>
<td>inclusive_scan_by_key(C,I,min)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>// total number of elements in previous segments</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>transform(H+J)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>L</td>
<td>scatter(E,K)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
### KD Tree: renumberRanks (further segmented)

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input ranks</td>
<td>0 1 2 3 1 0 2 3</td>
</tr>
<tr>
<td>B</td>
<td>Input flags</td>
<td>F F T T F F T T</td>
</tr>
<tr>
<td>C</td>
<td>Input segmentIds</td>
<td>0 0 1 1 2 2 3 3</td>
</tr>
<tr>
<td>D</td>
<td>Input pre-split segmentIds</td>
<td>0 0 0 0 1 1 1 1</td>
</tr>
<tr>
<td>E</td>
<td>CountingIterator(0)</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>F</td>
<td>inclusive_scan_by_key(D,E,min)</td>
<td>0 0 0 0 4 4 4 4</td>
</tr>
<tr>
<td>G</td>
<td>transform(A+F)</td>
<td>0 1 2 3 5 4 6 7</td>
</tr>
<tr>
<td>H</td>
<td>ConstantIterator(1)</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>I</td>
<td>exclusive_scan_by_key(C,H)</td>
<td>0 1 0 1 0 1 0 1</td>
</tr>
<tr>
<td>J</td>
<td>scatter(I,G)</td>
<td>0 1 0 1 1 0 0 1</td>
</tr>
<tr>
<td>K</td>
<td>scatter(B,G)</td>
<td>F F T T F F T T</td>
</tr>
<tr>
<td>L</td>
<td>segmentedSplit(J,K,C)</td>
<td>0 1 0 1 1 0 0 1</td>
</tr>
<tr>
<td>M</td>
<td>inclusive_scan_by_key(C,E,min)</td>
<td>0 0 2 2 4 4 6 6</td>
</tr>
<tr>
<td>N</td>
<td>transform(L+M)</td>
<td>0 1 2 3 5 4 6 7</td>
</tr>
<tr>
<td>O</td>
<td>scatter(I,N)</td>
<td>0 1 0 1 1 0 0 1</td>
</tr>
</tbody>
</table>
ADDITIONAL DETAILS ON RESEARCH RESULTS
Distributed Data Parallelism

- Goal: Allow developer to write data-parallel algorithms in the same way whether vectors are actually stored on one node or multiple distributed nodes

- Related work
  - Guy Blelloch: data parallel algorithms, including nested data parallelism using segment descriptors
  - Bergstrom and Reppy: port Blelloch’s NESL language to GPUs
  - Strengert, Muller, Dachsacher, Ertl, Frey: CUDASA extends CUDA to run across multiple GPUs, presenting distributed shared memory to developer as global arrays
  - OpenMPI and OpenRTE: improve MPI to run well across large heterogeneous systems
  - Libraries for writing parallel code: Thrust, STAPL, Dax, EAVL, DIY

- Our contribution: Portable performance on-node and across nodes with same easy-to-use STL-like API, supporting many algorithms including use of nested parallelism
Distributed Data-Parallel Primitives

- Created a distributed wrapper for Thrust which handles the communication in the backend using MPI while still calling the standard Thrust library to take advantage of available on-node parallelism.
- For example, call `dthrust::scan(input.begin(), input.end(), output.begin())` instead of `thrust::scan(input.begin(), input.end(), output.begin())`.
- Operators implemented:
  - Data Transfer Functions (host to device and device to host)
  - Rebalance and Shift
  - Transform, Scan, Segmented Scan, Scatter, Gather, Reduce
  - Upper bound (binary search), sort, sort by key
Distributed Scan

```
<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
```

thrust::inclusive_scan

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>11</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>13</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>15</td>
<td>18</td>
</tr>
</tbody>
</table>
```

MPI_Gather

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14</td>
<td>12</td>
<td>20</td>
</tr>
</tbody>
</table>
```

thrust::inclusive_scan

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14</td>
<td>26</td>
<td>46</td>
</tr>
</tbody>
</table>
```

MPI_Scatter

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14</td>
<td>26</td>
<td>46</td>
</tr>
</tbody>
</table>
```

thrust::transform

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>16</td>
<td>21</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>29</td>
<td>37</td>
<td>39</td>
<td>46</td>
<td>55</td>
<td>57</td>
<td>61</td>
<td>64</td>
</tr>
</tbody>
</table>
```

Alternative: MPI_Scatter
Code for Distributed Scan Backend

```cpp
template <typename inputIterator, typename outputIterator, typename T, typename BinaryOperation>
OutputIterator scan(inputIterator first, inputIterator last, outputIterator result, T init, bool inclusiveScan, BinaryOperation binop, bool reverse)
{
    // Get the MPI rank, total number of processes, MPI data type, and local vector size
    int commSize; MPI_CHECK(MPI_Comm_size(MPI_COMM_WORLD, &commSize));
    int commRank; MPI_CHECK(MPI_Comm_rank(MPI_COMM_WORLD, &commRank));
    MPI_Datatype dataType;
    int dataTypeFactor;
    dthrust::get_mpi_type<T>(typeid(T), dataType, dataTypeFactor);
    int N = last - first;

    // Perform the inclusive or exclusive scan locally on this processor
    if (inclusiveScan) dthrust::inclusive_scan(first, last, result, binop);
    else if (commRank == 0) dthrust::exclusive_scan(first, last, result, init, binop);
    else dthrust::exclusive_scan(first, last, result, 0, binop);

    // Get the local sum (the last element of the scan, combined with the last element of the input in case of an exclusive scan)
    T localSum = init;
    if (N > 0) localSum = *(result+N-1);
    if (!((inclusiveScan) && (N > 0))) localSum = binop(localSum, *(last-1));

    // Gather local sums to the root, perform a scan on those, and scatter these offsets
    dthrust::host_vector<T> localSums;
    if (commRank == 0) { localSums.resize(commSize+1); dthrust::fill(localSums.begin(), localSums.end(), init); }
    MPI_CHECK(MPI_Gather(&localSum, 1, dataType, dthrust::raw_pointer_cast(&localSums.begin()) + (reverse ? 0 : 1)), 1, dataType, 0, MPI_COMM_WORLD);
    if (commRank == 0) if (reverse) dthrust::inclusive_scan(localSums.rbegin(), localSums.rend(), localSums.rbegin(), binop);
    else dthrust::inclusive_scan(localSums.begin(), localSums.end(), localSums.begin(), binop);
    MPI_CHECK(MPI_Scatter(dthrust::raw_pointer_cast(&localSums.begin()) + (reverse ? 1 : 0)), 1, dataType, &localSum, 1, dataType, 0, MPI_COMM_WORLD);

    // Apply the received offset to each element on this processor
    dthrust::transform(result, result+N, dthrust::make_constant_iterator(localSum), result, binop);
    return (result+N);
}
```
Algorithms Built Using the Distributed Primitives

- **Isosurface**
  - Same algorithm as in our EGPGV paper, generating a “reverse map” from output vertex index to input cell index, allowing it to “lazily” apply operations only to cells that will generate the output vertices
  - Generates “triangle soup”, but generated vertices could potentially be welded into a triangle strip using data-parallel algorithm in Thrust weld_vertices example
  - Communication overhead is low, but load may not be balanced

- **KD-tree construction**
  - Generally based on algorithm from Blelloch’s dissertation, but tailored to Thrust API
  - Uses nested data parallelism, creating sub-trees top-down
  - Points may have to be moved to a different processor in first $\log_2(p)$ levels
Code for Distributed Isosurface Algorithm

```cpp
// Initialize a counting iterator, and allocate memory for vectors
thrust::counting_iterator<int> countingIterator; dthrust::make_counting_iterator(0, input.NCells, countingIterator);
dthrust::resize(input.NCells, case_indices); dthrust::resize(input.NCells, num_vertices); dthrust::resize(input.NCells, valid_cell_enum);

// Classify all cells to get Marching Cubes case index and number of vertices to generate
thrust::transform(countingIterator, countingIterator+case_indices.size(), thrust::make_zip_iterator(thrust::make_tuple(
case_indices.begin(), num_vertices.begin()), classify_cell(input, isovalue, numVertTable.begin())));

// Enumerate the valid cells, and then find the indices of the valid cells
dthrust::transform_inclusive_scan(num_vertices.begin(), num_vertices.end(), valid_cell_enum.begin(), is_valid_cell, thrust::plus<int>())
dthrust::upper_bound_counting(valid_cell_enum.begin(), valid_cell_enum.end(), num_valid_cells-1, valid_cell_indices);

// Use valid cell indices to fetch case index and number of vertices for each valid cell
dthrust::gather(num_vertices.begin(), num_vertices.end(), valid_cell_indices.begin(), valid_cell_indices.end(), output_vertices_compact);
dthrust::gather(case_indices.begin(), case_indices.end(), valid_cell_indices.begin(), valid_cell_indices.end(), case_indices_compact);

// Do an enumeration to get output indices for first vertex generated by valid cells
output_vertices_enum.resize(output_vertices_compact.size());
dthrust::exclusive_scan(output_vertices_compact.begin(), output_vertices_compact.end(), output_vertices_enum.begin(), 0, thrust::plus<int>())

// Get global and local number of vertices, and allocate space for vertex arrays
num_total_vertices = dthrust::back(output_vertices_compact) + dthrust::back(output_vertices_enum);
int num_local_vertices = 0;
if ((output_vertices_compact.size() > 0) && (output_vertices_enum.size() > 0))
    num_local_vertices = output_vertices_compact.back() + (output_vertices_enum.back() - output_vertices_enum.front());
vertices.resize(num_local_vertices); normals.resize(num_local_vertices); scalars.resize(num_local_vertices);

// Do edge interpolation for each valid cell
if (num_local_vertices > 0)
    thrust::for_each(thrust::make_zip_iterator(thrust::make_tuple(valid_cell_indices.begin(), output_vertices_enum.begin()),
case_indices_compact.begin(), output_vertices_compact.begin()), thrust::make_zip_iterator(thrust::make_tuple(valid_cell_indices.end(), output_vertices_enum.end), case_indices_compact.end(), output_vertices_compact.end())), isosurface_function(input, source, isovalue, output_vertices_enum[0], triable.begin(), thrust::raw_pointer_cast(&vertices.begin()), thrust::raw_pointer_cast(&normals.begin()), thrust::raw_pointer_cast(&scalars.begin()));
```

Los Alamos National Laboratory
Operated by Los Alamos National Security, LLC for the U.S. Department of Energy’s NNSA

LA-UR-14-20028
KD Tree: Overview

<table>
<thead>
<tr>
<th>Segment</th>
<th>Point Ids</th>
<th>X Ranks</th>
<th>Y Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>0 1 2 3 4 5 6 7</td>
<td>1 6 0 2 7 3 4 5</td>
<td>0 5 2 3 7 6 4 1</td>
</tr>
<tr>
<td>computeGlobalRanks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>computeFlags(X)</td>
<td>F T F F T F T</td>
<td>F T F F T F T</td>
<td>F T F F T F T</td>
</tr>
<tr>
<td>segmentedSplit</td>
<td>0 2 3 5 1 4 6 7</td>
<td>1 0 2 3 6 7 4 5</td>
<td>0 2 3 6 5 7 4 1</td>
</tr>
<tr>
<td>flags</td>
<td>F F F F T T T</td>
<td>F F F F T T T</td>
<td>F F F F T T T</td>
</tr>
<tr>
<td>segmentIds</td>
<td>0 0 0 0 1 1 1 1</td>
<td>0 0 0 0 1 1 1 1</td>
<td>0 0 0 0 1 1 1 1</td>
</tr>
<tr>
<td>renumberRanks</td>
<td></td>
<td>1 0 2 3 2 3 0 1</td>
<td>0 1 2 3 2 3 1 0</td>
</tr>
<tr>
<td>computeFlags(Y)</td>
<td>F F T T T F F F</td>
<td>F F T T T F F F</td>
<td>F F T T T F F F</td>
</tr>
<tr>
<td>segmentedSplit</td>
<td>0 2 3 5 6 7 1 4</td>
<td>1 0 2 3 0 1 2 3</td>
<td>0 1 2 3 1 0 2 3</td>
</tr>
<tr>
<td>flags</td>
<td>F F T T F F T T</td>
<td>F F T T F F T T</td>
<td>F F T T F F T T</td>
</tr>
<tr>
<td>segmentIds</td>
<td>0 0 1 1 2 2 3 3</td>
<td>0 0 1 1 2 2 3 3</td>
<td>0 0 1 1 2 2 3 3</td>
</tr>
<tr>
<td>renumberRanks</td>
<td></td>
<td>1 0 0 1 0 1 0 1</td>
<td>0 1 0 1 1 0 0 1</td>
</tr>
</tbody>
</table>
KD Tree: Segmented Split

A Input pointIds
B Input flags
C Input segmentIds
D exclusive_scan_by_key(C,B)
   // total number of true flags preceding in segment
E CountingIterator(0)
F inclusive_scan_by_key(C,E,min)
   // total number of elements in previous segments
G inclusive_scan_by_key(C,inverse(B))
   // total number of false flags so far in segment
H Reverse inclusive_scan_by_key(C,G,max)
   // total number of false flags in segment
I transform(I[i]=if(B[i]) F[i]+H[i]+D[i] else F[i]+G[i]-1))
   // if true, new index is total number in previous segments
   // plus total number of false in this segment plus number of
   // trues preceding it in the segment; if false, new index is
   // total number in previous segments plus number of false
   // preceding it in the segment
J scatter(A,I)

0 2 3 5 1 4 6 7
FFTT TFF
0000 1111
0001 0122
0123 4567
0000 4444
1222 0012
2222 2222
0123 6745
0235 6714
KD Tree: Renumber Ranks
Isosurface Weak Scaling on CPUs

3D Isosurface Generation: OpenMP Compute Rates on four Intel Xeon E5-2670 nodes on Moonlight supercomputer
Isosurface Strong Scaling on CPUs

3D Isosurface Generation: Scaling with the number of Intel Xeon E5-2670 nodes on Moonlight supercomputer
Isosurface Strong Scaling with Threads

3D Isosurface Generation: Scaling with on-node CPU parallelism on four Intel Xeon E5-2670 nodes on Moonlight supercomputer
Isosurface Weak Scaling on GPUs

3D Isosurface Generation: CUDA Compute Rates on four NVIDIA Tesla M2090 GPUs on Moonlight supercomputer
Isosurface Strong Scaling on GPUs

3D Isosurface Generation: Scaling with the number of NVIDIA Tesla M2090 GPUs on Moonlight supercomputer
Isosurface of Ocean Temperature Data
Generated Across Four GPUs

Isosurface at 10°C computed across four NVIDIA Quadro 5000 GPUs on our Darwin cluster on a 3600x2400x42 ocean temperature data set.
KD Tree Strong Scaling with Threads

3D KD Tree Construction: Strong Scaling with OpenMP Thread Count

3D KD Tree Construction: Scaling with the number of OpenMP threads on four AMD Opteron 6168 nodes on Darwin Cluster
KD Tree Strong Scaling on CPUs

3D KD Tree Construction: Scaling with the number of AMD Opteron 6168 nodes on Darwin Cluster
Conclusions

- Different kinds of algorithms can be implemented using this distributed data-parallel API, with a level of parallel efficiency commensurate with the nature of the problem
  - Good scaling with number of nodes or GPUs and with on-node parallelism (e.g., OpenMP threads) achieved for isosurface, very similar to manually partitioning the input or to using Parallel VTK
  - KD-tree algorithm shows more complex algorithms, including those with nested parallelism, can also be implemented in this model, but there is still a trade-off between increased time to move points to correct processors in first \( \log_2(p) \) levels and computational savings at lower levels

- On-going and Future Work
  - Distributed implementation of short materials science Ginzburg-Landau simulation for Exascale Co-Design Center for Materials in Extreme Environments
  - Layered backends to address other challenges in “exascale era” computing (I/O, fault tolerance, etc.)
PINION Project

Physics-Based Data Models and Architecture-Optimized Backends for a Portable Data-Parallel Computation Library

Christopher Sewell, CCS-7
Li-ta Lo, CCS-7
Marianne Francois, CCS-2
Jim Ahrens, CCS-7
The Goal: Vision 2015 (1/3)

- What we will provide
  - Data structures that allow scientists to program in a way that maps easily to the problem domain rather than dealing directly with 1D host/device vectors
  - Operators that provide data-parallel implementations of analysis and computational functions often used in physics simulations
  - Backends that optimize implementations of data parallel primitives for one or two emerging supercomputer hardware architectures
The Goal: Vision 2015 (2/3)

- How we will provide it
  - Open-source release, similar to our PISTON release
  - Fully functional, well-documented application for a specific hydrodynamics problem
The Goal: Vision 2015 (3/3)

- What domain scientists will then be able to do
  - Perform physics simulations for a wide range of applications that use a 3D grid (e.g. solid mechanics, astrophysics, and climate modeling)
  - Reuse our analysis and computation operators
  - Write their own operators using our high-level data model
  - Modify the example application code according to their problem definition
  - Run the same code on multiple current and next-generation supercomputing architectures, getting good parallel performance on each