Title: Data-Parallel Programming With PISTON and PINION

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

Intended for: Programming Models and Applications Meeting, Sandia National Laboratory, August 2014
Data-Parallel Programming With PISTON and PINION

Chris Sewell
Li-ta Lo
Marianne Francois
James Ahrens
Los Alamos National Laboratory
Advantages of High-Level Parallel Programming

- Supercomputer Hardware Advances Everyday
  - More and more 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
Blelloch’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

http://www.cs.cmu.edu/~blelloch/papers/Ble90.pdf
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
Simple Examples with Thrust Pseudocode

- Generate
  \[ \text{thrust::sequence(0,4)} \rightarrow 0 \ 1 \ 2 \ 3 \ 4 \]

- Transform
  \[ \text{input} \rightarrow 4 \ 5 \ 2 \ 1 \ 3 \]
  \[ \text{thrust::transform(+1)} \rightarrow 5 \ 6 \ 3 \ 2 \ 4 \]

- Compact
  \[ \text{input} \rightarrow 4 \ 5 \ 2 \ 1 \ 3 \]
  \[ \text{thrust::copy_if(even)} \rightarrow 4 \ 2 \]

- Expand
  \[ \text{input} \rightarrow 4 \ 5 \ 2 \ 1 \ 3 \]
  \[ \text{thrust::for_each(x,2x)} \rightarrow 4 \ 8 \ 5 \ 10 \ 2 \ 4 \ 1 \ 2 \ 3 \ 6 \]

- Aggregate
  \[ \text{input} \rightarrow 4 \ 5 \ 2 \ 1 \ 3 \]
  \[ \text{thrust::reduce(+)} \rightarrow 15 \]
Generate Data in Parallel

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

- A sequence of increasing or decreasing values
  
  - // Allocate space for ten integers, uninitialized
  
  ```cpp
  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: 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());

x: 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
y: 0.0 0.01 0.04 0.09 0.16 0.25 0.36 0.49 0.64 0.81 1.0
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: 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 & 0
  \end{bmatrix}
  \begin{bmatrix}
  1 & 1 \\
  1 & 0
  \end{bmatrix}
  \begin{bmatrix}
  1 & 1 \\
  1 & 0
  \end{bmatrix}
  \begin{bmatrix}
  1 & 1 \\
  1 & 0
  \end{bmatrix}
  
  \begin{bmatrix}
  1 & 1 \\
  1 & 0
  \end{bmatrix}
  \begin{bmatrix}
  2 & 1 \\
  1 & 1
  \end{bmatrix}
  \begin{bmatrix}
  3 & 2 \\
  2 & 1
  \end{bmatrix}
  \begin{bmatrix}
  5 & 3 \\
  3 & 2
  \end{bmatrix}
  \begin{bmatrix}
  8 & 5 \\
  5 & 3
  \end{bmatrix}
  
  \]
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

2  3  4  5  6  7  8  9 10 11 12 13 14 15
2  3  5  7  9 11 13 15
2  3  5  7 11 13
2  3  5  7 11 13
2  3  5  7 11 13
2  3  5  7 11 13
2  3  5  7 11 13
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
  - 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
Isosurface with Marching Cubes Algorithm

1. input
   transform(classify_cell)
2. caseNums

3. numVertices
   transform_inclusive_scan(is_valid_cell)
4. validCellEnum

5. CountingIterator
   upper_bound
6. validCellIndices

7. numVerticesCompacted
   exclusive_scan
8. numVerticesEnum
   for_each(isosurface_functor)
9. outputVertices

# of valid cells = 4
Total # of vertices = 10
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
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 CoGLGinzburg-Landau simulation

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

LA-UR-14-26186
Distributed Memory Parallelism

- 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
  
- Distributed Scan Algorithm

Isosurface of 3600x2400x42 ocean temperature data computed on 4 GPUs
New Data-parallel Algorithms Accelerate Cosmology Data Analysis on GPUs

Objectives

Milestone
- Implement application-specific visualization and/or analysis operators needed for in-situ use by LCF science codes
- Use PISTON to take advantage of multi-core and many-core technologies

Target Application
- The Hardware/Hybrid Accelerated Cosmology Code (HACC) simulates the distribution of dark matter in the universe over time
- An important and time-consuming analysis function within this code is finding halos (high density regions) and the centers of those halos

Impact

VTK-m framework
- The PISTON component of VTK-m develops data-parallel algorithms that are portable across many-core architectures for use by LCF codes
- PISTON consists of a library of visualization and analysis algorithms implemented using Thrust, and our extensions to Thrust

Halo and Center Finders
- Data-parallel algorithms for halo and center finding implemented using VTK-m (PISTON) allow the code to take advantage of parallelism on accelerators such as GPUs
- Can be used for post-processing or in-situ, with in-situ integration directly into HACC or via the CosmoTools library

Accomplishments

Performance Improvements
- On Moonlight with 1024^3 particles on 128 nodes with 16 processes per node, PISTON on GPUs was 4.9x faster for halo + most bound particle center finding
- On Titan with 1024^3 particles on 32 nodes with 1 process per node, PISTON on GPUs was 11x faster for halo + most bound particle center finding
- Portability of PISTON allowed us to also run our algorithms on an Intel Xeon Phi
- Implemented grid-based most bound particle center finder using a Poisson solver that performs fewer total computations than standard O(n^2) algorithm

Science Impact
- These performance improvements allowed halo analysis to be performed on a very large 8192^3 particle data set across 16,384 nodes on Titan for which analysis using the existing CPU algorithms was not feasible

Publications
- Submission: “Utilizing Many-Core Accelerators for Halo and Center Finding within a Cosmology Simulation” Christopher Sewell, Li-ta Lo, Katrin Heitmann, Salman Habib, and James Ahrens

Visual comparison of halos computed by the original HACC algorithms (left) and the PISTON algorithms (right). The results are equivalent, but are computed much more quickly on the GPU using PISTON.
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
Selected Mesh Operators

**vertex_to_edges_op** Adjacency operator for vertices, given one vertex id, return ids of 4 edges sharing the vertex as \{Left, Right, Bottom, Top\}, -1 means non-existence/boundary edges.

**vertex_to_cells_op** Adjacency operator for vertices, given one vertex id, return ids of 4 cells sharing the vertex as \{Lower Left, Lower Right, Upper Left, Upper Right\}, -1 means non-existence/boundary cells.

**edge_to_vertices_op** Boundary operator for edges, given one edge id, return ids of the two end vertices as \{Left, Right\} or \{Bottom, Top\}.

**edge_to_cells_op** Coboundary/adjacency operator for edges, given one edge id, return ids of 2 cell ids sharing the edge.

**cell_to_edges_op** Boundary operator for cells, given one cell id, return ids of 4 edges as \{Bottom, Right, Top, Left\}.

**cell_to_vertices_op** Second order boundary operator for cells, given one cell id, return ids of the 4 vertices as \{Lower Left, Lower Right, Upper Left, Upper Right\}.

**cell_von_neumann_neighbor_op** Given a cell return the 4 orthogonal neighboring cells in the following order \{West, East, South, North\}.

**cell_moore_neighbor_op** Given a cell return the 8 neighboring cells in the following order \{W, E, S, N, SW, SE, NW, NE\}.

**vertex_position_op** Given a vertex id, return the coordinates of the position of that vertex.

**cell_center_position_op** Given a cell id, return the coordinates of the cell center position of that cell.

**edge_center_position_op** Given an edge id, return the coordinates of the edge center position of an edge.

**edge_normal_op** Given an edge id, return the orthogonal vector (i.e. the normal) to that edge. The direction of the normal vector always points to the "right" side of the edge. The magnitude of the vector is the length of the edge.
Volume Fraction Initialization

- **Description**
  - Given computational mesh and mathematical expression of a shape (circle, square, line,...), compute the volume fractions in every cell
    - \( \text{Vof}=0 \) cell is empty
    - \( \text{Vof}=1 \) cell is full
    - Divide and conquer algorithm
      - Recursive algorithm
      - Check if vertices are in or out
      - Refine cells up to a given lowest level
    - Implemented in 1D, 2D and 3D

```cpp
    thrust::transform(grid.cell_id_begin(), grid.cell_id_end(),
                      d_vof.begin(),
                      make_vof_init(grid, circle()));
```

Resulting volume fractions for a circle of radius 0.25 centered at (0.5,0.5) in a unit square. Mesh size is 10x10.
Interface Reconstruction: Method

- Piecewise Linear Interface Calculation (PLIC) algorithm
  - Given volume fractions, compute the interface normal, as the gradient of the volume fraction using Green-Gauss
  - Then, find the line equation that intersects the computational cell
  - Use of a lookup table to identify intersection case
  - At each iteration compute polygon area
  - Iterate until polygon area match the area given by the cell volume fraction


Lookup table and 2 examples

```c
const int tessellation_table[16][8] = {
    // nb. of cell vertices, cell vertices, cell edges
    { 0, -1, -1, -1, -1, -1, -1, -1 },
    { 1, 0, 1, 0, 2, -1, -1, -1 },
    { 1, 1, 1, 3, 0, 1, -1, -1 },
    { 2, 0, 1, 3, 0, 2, -1, -1 },
    { 1, 2, 0, 2, 3, 3, 3, 1 },
    { 2, 2, 0, 0, 1, 2, 3, -1 },
    { 0, -1, -1, -1, -1, -1, -1, -1 },
    { 3, 2, 0, 1, 1, 3, 2, 3 },
    { 1, 3, 2, 3, 1, 3, -1, -1 },
    { 0, 1, -1, -1, -1, -1, -1, -1 },
    { 2, 1, 3, 2, 3, 0, 1, -1 },
    { 3, 0, 1, 3, 2, 0, 2, -1 },
    { 2, 3, 2, 0, 2, 1, 3, -1 },
    { 3, 3, 2, 0, 0, 1, 1, 3 },
    { 3, 1, 3, 2, 0, 2, 0, 1 },
    { 0, -1, -1, -1, -1, -1, -1, -1 }
};
```
Interface Reconstruction: Results

Resulting interface reconstruction planes and interface normals for the case of a circle of radius 0.25 centered at (0.5,0.5) in a unit square. Mesh size is 10x10.

Movie shows iterative procedure for finding intersection plane that matches the volume fractions.
Weak scaling – Interface reconstruction

Weak scaling plots for the interface reconstruction (iterative volume matching procedure) algorithm. The grid sizes range from $256^2$ to $8192^2$. 
Strong scaling obtained on the MIC for a constant grid size of \(1,024^2\)
Conclusions

- Data-parallel programming: get portable, parallel code by implementing using a few data-parallel primitives
- PISTON: data-parallel visualization and analysis algorithms
- PINION: data-parallel simulation codes