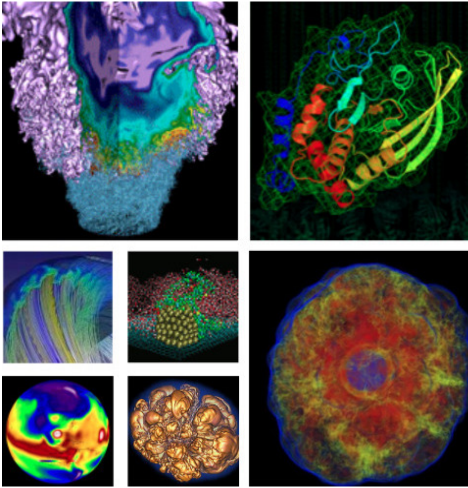
TomoPy
A CUDA Case Study

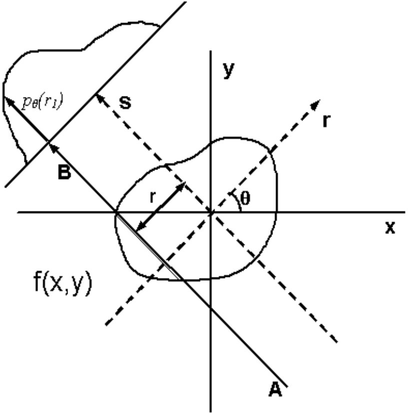


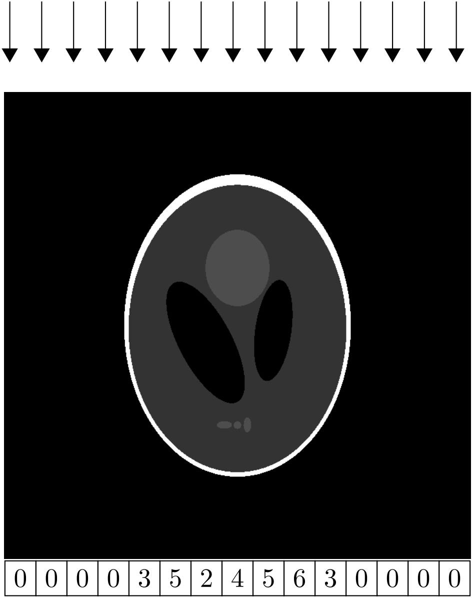
National Energy Research Scientific Computing Center

Jonathan R. Madsen, PhD
✉ jrmadsen@lbl.gov
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory

July 2, 2019

- Tomographic reconstruction is a multidimensional inverse problem where the challenge is to yield an estimate of a specific system from a finite number of projections.
- Heavily used technique at light sources for structural imaging of materials samples and biological specimens at high-resolution.
- A series of rays such as **B** are passed through a sample and $p_\theta(r_1)$ is recorded back side (projection)
- In general, a reconstruction starts with an array of projection angles and the array of projection values at each projection angle and simulates the imaging in reverse.



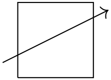

0 0 0 0 3 5 2 4 5 6 3 0 0 0 0

Listing 1: General reconstruction workflow

```
for 0 : num_iterations          # 1 - 500+
    for 0 : num_slices          # 1 - 1000+
        for 0 : num_angles      # 360 - 1500+
            for 0 : num_pixels  # 512 - 2048+
                do_calculation(...)
```

- Loop over iterations is order-dependent
- Loop over slices is fully independent
- Loop over projection angles is fully independent (for target algorithms)
- Loop over number of pixels is conditionally independent
    - When projection angles are processed in parallel, updating pixels can become data-race

- A pool of threads is introduced at the Python-level per-slice
  - Perfect scaling w.r.t. # of slices
- Calculates the traversal distance through the pixels at the given projection angle and offset from center as a weighting factor



- The value of the projection is “distributed” along all the intersecting pixels according to the calculated weighting factor

- This algorithm required several supplementary arrays for each iteration of the projection angles

```
// arrays of intersection points
float* ax     = (float*) malloc((ngridx + ngridy) * sizeof(float));
float* ay     = (float*) malloc((ngridx + ngridy) * sizeof(float));
float* bx     = (float*) malloc((ngridx + ngridy) * sizeof(float));
float* by     = (float*) malloc((ngridx + ngridy) * sizeof(float));
// sorted intersection points
float* coorx  = (float*) malloc((ngridx + ngridy) * sizeof(float));
float* coory  = (float*) malloc((ngridx + ngridy) * sizeof(float));
// distances between intersection points and index mapping
float* dist   = (float*) malloc((ngridx + ngridy) * sizeof(float));
int*   indi   = (int*)   malloc((ngridx + ngridy) * sizeof(int));
```

- Common optimizations were implementated

1. Minimize data transfers
2. Introduced streams
3. Block and grid size optimizations

- Significant progress was achieved w.r.t. original GPU implementation but was still slightly slower than CPU
- "Sorting" and "trimming" were significant bottlenecks ⇒ consumed 95% of run-time
- Memory access was inherently strided in a main kernel (and atomic op)

```
for(int n = 0; n < csize - 1; ++n)
  data[d + p*dx + s*dt*dx] += model[indi[n] + s*ry*rz] * dist[n];
```

- Given the relatively similar compute times on CPU vs. GPU, a secondary thread-pool was introduced per "Python" thread to handle large data sets with 1,000+ slices
  - The idea here was to increase parallelism and further sub-divide the work between the CPU and GPU ⇒ use GPU to supplement CPU when exceeding # of cores
  - If GPU began to out-perform CPU ⇒ offload to GPU until OOM and the threads would fall back to CPU

- Summary: optimizing an algorithm that was designed for the CPU

- TomoPy lead noted there was a rotation-based technique no longer used in tomography (for performance reasons) that removed the sorting and trimming requirements and where all the weight became 1
- Rotation-based method was computationally expensive:
  - Rotated the entire ROI to be parallel with the incident ray
  - interpolated the pixels to their new coordinates
  - Required padding the projections (*i.e.*, larger reconstruction) to account for pixel loss during rotation
- In addition to removing the sorting and trimming bottlenecks, the method *also* aligned the memory access
- In other words, there was an alternative algorithm that was more computationally expensive and increased the problem size but *removed our parallelism bottlenecks...*

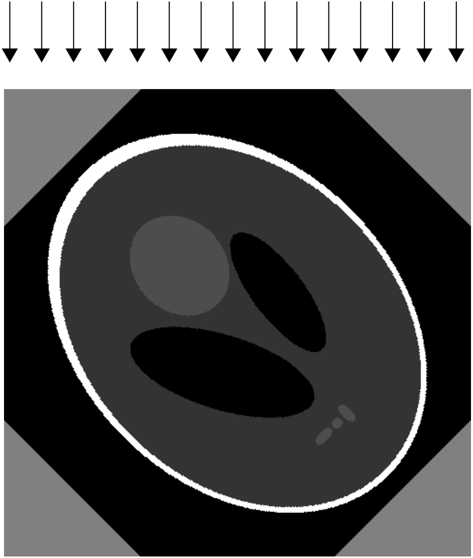


Figure 1: Reconstructed image is shown for demonstration purposes

Final GPU implementation

- The new algorithm essentially turned the per-projection workflow into:
  1. Rotate ROI by $-\theta$
  2. Distribute projection value along a row of pixel values in ROI
  3. Rotate ROI by $+\theta$
  4. Update reconstruction
- Each thread started at the Python level is assigned a device in round-robin fashion:

```
int num_devices;
cudaGetDeviceCount(&num_devices);
static std::atomic<int> thread_counter;
cudaSetDevice((thread_counter++) % num_devices);
```

- Instead of creating a pool of CUDA streams for the parallel loop over projection angles, the secondary thread-pool was retained $\Rightarrow$ each thread in secondary pool created one CUDA stream
  - *e.g.*, Instead of 1 thread with 12 streams $\Rightarrow$ 12 threads with 1 stream

- Implementation tip: Host threads can be used in lieu of explicit CUDA streams in certain situation
    - NVCC compiler flag "--default-stream per-thread" will cause the default stream (0) to be asynchronous w.r.t. other host threads but may not propagate to external library calls
    - Replace `cudaDeviceSynchronize()` with `cudaStreamSynchronize(0)`
- The formerly discarded algorithm became a quintessential example of why GPUs were created in the first place
- Recorded performance numbers w.r.t. Edison supercomputer: 50 iterations, 1-24 slices, 1500 projections angles, 2048 pixels
    - Edison node with 24 threads started at Python level
    - Cori-GPU (V100) node with 8 GPUs and one "Python" thread per GPU each with ~12-24 secondary threads/streams
- New algorithm introduced interpolation methods: nearest-neighbor, linear, cubic

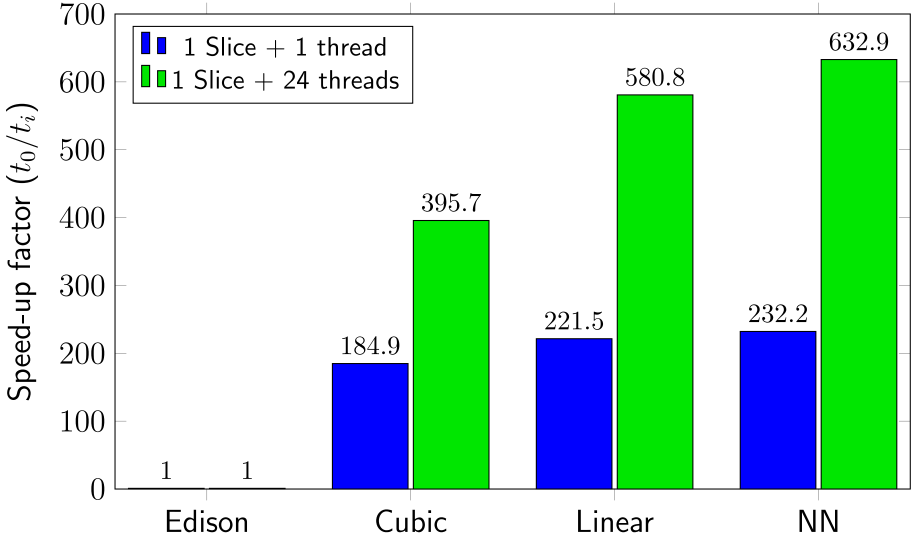


Figure 2: TomoPy single-slice speed-up with various tasking threads on Cori-GPU nodes w.r.t. TomoPy v1.2

Table 1: Single-slice reconstruction times ($22594.2\text{sec} \approx 6.25\text{hr}$)

Machine	Method	# Thread	Wall time (sec)	Speed-up
Edison	Ray	1	22594.2	1
Cori-GPU	Cubic	1	122.2	184.9
Cori-GPU	Linear		102.0	221.5
Cori-GPU	NN		97.3	232.2
Cori-GPU	Cubic	24	57.1	395.7
Cori-GPU	Linear		38.9	580.8
Cori-GPU	NN		35.7	632.9

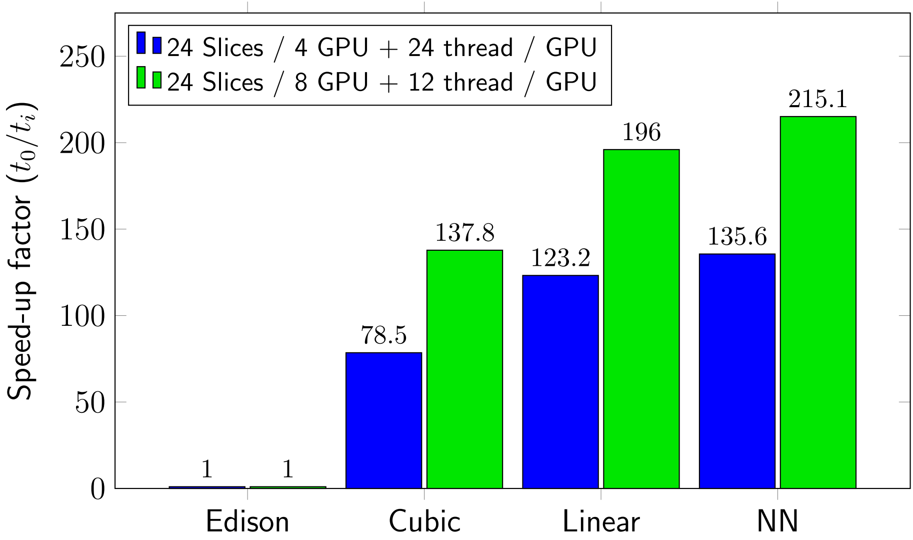


Figure 3: TomoPy full node speed-up with 4 and 8 GPUs (96 total threads) on Cori-GPU nodes w.r.t. TomoPy v1.2

Table 2: Scientific throughput reconstruction times ($22951.8\text{sec} \approx 6.35\text{hr}$)

Machine	Method	# GPU	Wall time (sec)	Speed-up
Edison	Ray	0	22951.8	1
Cori-GPU	Cubic	4	292.3	78.5
Cori-GPU	Linear		186.3	123.2
Cori-GPU	NN		169.2	135.6
Cori-GPU	Cubic	8	166.5	137.8
Cori-GPU	Linear		117.1	196.0
Cori-GPU	NN		106.7	215.1

- The secondary thread-pool concept was retained from first developments based on the idea that:
  1. Submitting work to GPU reduced to a large (serial) loop launching kernels
  2. Individual CPU cores on HPC machines operate at a low frequency
     $\Rightarrow$ serial performance is much slower
  3. Synchronization on the GPU does not require CPU cycles
     $\Rightarrow$ over-subscribe the # of threads relative to the # of CPU cores
  4. Amdahl's law which states the theoretical speed-up from parallelism is restricted by the serial portions of the workload
- In the end though, these benefits did not appear to show up at scale
  - Subsequent analysis of the CPU time indicated the threads were very busy
    $\Rightarrow$ a potential indicator of relevant work
  - "Under-the-hood", CUDA is implementing spin-mutexes at the synchronization step(s)
    $\Rightarrow$ artificially increasing the CPU time

1. Original CPU algorithm not well-suited for the GPU
2. Introduced an alternative algorithm that was more computationally expensive and increased the problem size that results in massive speed-up
   - **Don't be afraid to restructure the entire problem when there is the potential to reduce logic in exchange for FLOPS**
3. Multi-threading does not need to be removed when migrating to the GPU
4. When the algorithm runs entirely on the GPU, there is no discernible performance difference between using threads with a single stream and one thread with multiple streams
5. If you are planning to do hybrid CPU/GPU work, be wary of spin-mutexes and investigate the affect of setting device flags, *e.g.*, `cudaSetDeviceFlags(cudaDeviceScheduleSpin)` vs. `cudaSetDeviceFlags(cudaDeviceScheduleYield)`
   - Default is a heuristic based device flag `cudaDeviceScheduleAuto`