Parallelizing Lunar Safe Landing Algorithms on the Tilera Tile64 Processor

Carlos Villalpando¹, Steve Goldberg,²
Andrew E. Johnson¹, Raphael Some¹

POC: Carlos.Villalpando@jpl.nasa.gov
818-354-7487

[1] Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Dr.
Pasadena, CA 91109

ALHAT VISION STATEMENT
Develop and mature to TRL6 an autonomous lunar landing GN&C and sensing system for crewed, cargo, and robotic lunar descent vehicles. The System will be capable of identifying and avoiding surface hazards to enable a safe precision landing to within tens of meters of certified and designated landing sites anywhere on the Moon under any lighting conditions.
Hazard Detection and Avoidance (HDA) overview

Mosaic of lidar images generated using gimbal as spacecraft descends

Elevation map is constructed from lidar images

Safety map sent to AFM for selection of safe and reachable site

HDA algorithm detects slope and roughness hazards and computes safety map

Roughness Map

Slope Map
HDA Tasks Sequences

LIDAR Input

**Motion Correction**
Transform 3D shapes of LIDAR from Sensor to Ground frame

**Elevation Map Generation**
Project 3D samples into 2D elevation map

Compute Slope & Roughness

Compute Hazards

Compute Safe Site Selection
HDA on Tile64
HDA

Goals:
- Evaluate suitability of Tilera Tile64 to process LIDAR data for HDA application.
- Evaluate power vs. performance of ported algorithm
- Provide data to ALHAT for system optimization studies
- Optimize algorithm towards selected resource utilization.

Challenges:
- Original algorithm is single thread C++
- Original algorithm is memory intensive
- One algorithm component’s 'inner-loop' of FP operations alone used ~13s on a single Tile64 PE
General Approach

1. Use structured map to improve cache localization.
   - Map data is a 64 bit union of variable types
   - Input/Output map types depend on algorithmic needs
2. Where possible perform map operations in-place to reduce memory footprint and memory bandwidth.
   - All maps are only used once so next process can overwrite input with output.
4. Parallelize where possible.
Accumulate

Procedure
1. Take scan data from flash or scanning LIDAR
2. Interpolate spacecraft trajectory to match scan times
3. Transform each laser return from sensor to LSLF (Lunar Surface Local Frame)
4. Interpolate laser measurements into up to 4 map cells, each cell receiving a different weight and incremental sum.

Challenges
1. 65k laser measurements per 128x128 LIDAR flash scan
2. Output memory access is non-monotonic and potentially “random”
Accumulate
Accumulate

Approach
1. Structured map contains two 32 bit floating point values per pixel.
2. Parallelize by partitioning scan data into PE regions
   - Partitioning made best use of available cache

Samples placed in shared memory
Defines start and stop addresses in shared memory for each PE and sends parameter message to worker bees, who then iterate internally on the samples in the shared memory segment assigned to it

PE[0]
- Rotate/Translate
- Generate Weights
- Generate Sums
- Lock Map
- Update Map
- Release Map
- Loop

PE[1]
- Rotate/Translate
- Generate Weights
- Generate Sums
- Lock Map
- Update Map
- Release Map
- Loop

PE[N]
- Rotate/Translate
- Generate Weights
- Generate Sums
- Lock Map
- Update Map
- Release Map
- Loop

Barrier
Average

Procedure
1. Convert Accumulated Sums into Elevation map

Approach
1. Divide work among PEs by row.
2. Run in place

PE[0]
- Working Row = 0
- Compute Elevation
- Row = Row + #of PEs
- Loop

PE[1]
- Working Row = 1
- Compute Elevation
- Row = Row + #of PEs
- Loop

PE[N]
- Working Row = n
- Compute Elevation
- Row = Row + #of PEs
- Loop

Barrier
Grassfire

Procedure
1. Compute distance to nearest valid data
2. For each grassfire level requested, interpolate valid data

Challenges
1. Computing initial distances is 'connected components'
2. Computing initial distances requires forwards backwards processing
Grassfire

Grassfire fills in holes in elevation data.

Approach
1. PE 0 computes distance values, a connected components problem. Rest are idle.
2. All PEs compute interpolation value.
Slope and Roughness

Procedure
1. Compute slope and roughness over a window W for each valid map cell

Challenges
1. When given all 9 coefficients for each window, just the slope and roughness math takes $> 13$ seconds on single PE
   - $\sim100$ floating point operations + $\text{acos} + \text{sqrt} + \text{fabs}$/cell
2. Brute force computation of all 9 coefficients required $> W^2 W^9$ floating point operations

Approach
1. Use 2D sliding sums to reduce coefficients math to $\sim 45$ floating point operations
2. Divide sliding sums among 3 PE's
3. Use shared memory to store coefficients
   - Faster than per cell IPC
   - Faster and lower latency than end of row IPC
4. Remaining PE's compute slope and roughness via round robin scheduling
Compute Slope and Roughness

**Elevation Map**

**PE[0]**
- Compute row and column rolling sum for coefficients 1-3
- Update row segment status

**PE[1]**
- Compute row and column rolling sums for coefficients 4-6
- Update row segment status

**PE[2]**
- Compute Row and column rolling sums for coefficients 7-9
- Update row segment status

**Shared Memory Row Segment status and Coefficients 1-9**

**PE[3]**
- Col = 0
- Busy Loop wait
- Compute Slope and roughness from Coefficients 1-9
- Col += #PEs - 3
- Loop

**PE[N]**
- Col = #PEs -3
- Busy Loop wait
- Compute Slope and roughness from Coefficients 1-9
- Col += #PEs - 3
- Loop

**Row Barrier**
Hazard Detection

Procedure
1. Threshold Slope and Roughness to find hazards
2. Dilate and Erode hazard map
3. Invert hazard map to non-hazard map
4. Label connected non-hazardous regions
5. Measure and mark largest safe region

Challenges
1. Parallelize connected components

Approach
1. Structured map contains 2 16bit Booleans and 32bit area label
2. Parallelize Slope and Roughness thresholding
3. Parallelize Dilate and Erode
   o Each PE works on rows where row mod # of PEs = PE
   o Results are stored in temp memory until next PE is finished
   o IPC used at end of row to release previous PE
4. Parallelize inversion of hazard to non-hazardous
Map Obstacle Labeling

Slope & Roughness

PE[0]
  Working Row = 0
  Process Row
  Row += #of PEs
  Loop

PE[1]
  Working Row = 1
  Process Row
  Row += #of PEs
  Loop

PE[N]
  Working Row = n
  Process Row
  Row += #of PEs
  Loop

Barrier

Marks pixel as obstacle if slope and roughness threshold exceeded
Map Dilate

“No obstacle” map

PE[0]
Working Row = 0
Grow Object
Send msg to next
Recv msg from prev

Wait for msg

Msg received

Update Map
Row += #of PEs
Loop

PE[1]
Working Row = 1
Grow Object
Send msg to next
Recv msg from prev

Wait for msg

Msg received

Update Map
Row += #of PEs
Loop

PE[N]
Working Row = n
Grow Object
Send msg to next
Recv msg from prev

Wait for msg

Msg received

Update Map
Row += #of PEs
Loop

Barrier

Each PE waits until the next PE is finished before moving to the next Row
• Majority of time is spent in Slope and Roughness calculations

• Adding sample frames to map throughput increase is linear with # of PEs.
  • This function has many cache misses due to random memory accesses, though memory bottlenecking does not seem to be coming into play.

• Most other functions begin diminishing returns at 8-10 PEs.
What didn't work

- Message passing call per map cell
  - Attempted in Map Accumulate
  - Attempted in slope and roughness calculations
  - Reason: PE's spent all their time sending or receiving messages.
- Software caching of shared memory writes. Re-ordering writes to make memory access as monotonic as possible to minimize cache misses.
  - Tried insert sort, binary trees, address hashing of transformed scan data in accumulate
  - Reason: Time spent in re-ordering writes was equal or greater than cache miss penalty savings.
- 8 bit binary operations in dilate, erode and label
  - Compare/branch faster than binary operations
  - Reason: C standard allows for logical short-circuiting. Logical short circuit of A||B||C||D||E faster than bitwise A|B|C|D|E, then compare
- Cache blocking
  - Reordered 'for loops' to make better use of cache sets
  - Reason: Enough map lines already fit in cache.
What Else Can Be Attempted?

- Re-optimize Slope & Roughness from the current 3-5 PE split to a 4-12 PE split

- Parallelize region or grassfire labeling
  - Process is a “connected components” type problem and is difficult to parallelize. Taking lessons from FPGA implementations of connected components for ideas.

- Combine algorithms to reduce passes through shared memory
  - Decreases code readability but increases temporal locality.
Conclusions

1. Shared memory is faster than messaging when PE's are close enough to memory banks.
   - Messaging required explicit IPC calls

2. An IPC call per map cell is very inefficient, spin loop on shared memory objects performance was better than IPC.

3. Shared memory can be optimized with intelligent arranging of PEs used. A “square” array of PEs was better than “rectangular” array due to reduced Manhattan distances between PEs. The latency for setting up and tearing down a shared memory access and/or a message passing call is reduced with shorter distances between PEs