Efficient GPU Techniques for Processing Temporally Correlated Satellite Image Data

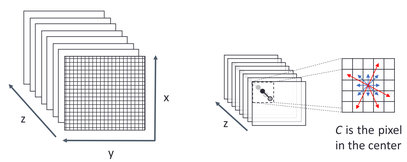
Tahsin Reza*, Tanuj Kr Aasawat *, Matei Ripeanu*
*Electrical and Computer Engineering
University of British Columbia, Vancouver

Dipayan Mukherjee *†
†Computer Science and Engineering
Indian Institute of Technology, Kharagpur, India

netsyslab.ece.ubc.ca

1. Problem and Contributions

- Processing of *spatio-temporal* data on massively parallel many-core platforms such as GPUs.
- Commonly used data structures, e.g., 3D arrays, incurs scattered data accesses
- Difficult to prevent branch divergence and ensure memory coalescing on GPU



- A stack of 2D images taken at different points in time
- *x* and *y* are the spatial axes
- *z* is the time axis
- (*x*, *y*, *z*) is the coordinate of a pixel in the stack
- Every pixel *C* in the stack couples up (forms an edge) with all the neighbours *F* (within a predefined distance) along the spatial dimensions and computes $u=f(C, N \in F)$ according to some function *f*
- Computes normalized *u* along the time axis

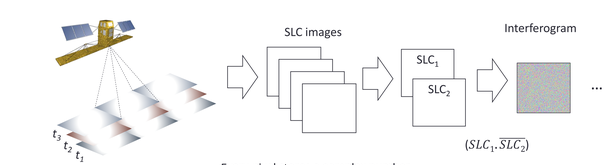
Contribution highlights

- CUDA implementation of a real-world application which is being used in production
- It is 18x faster on a single Tesla K40 GPU than on a 20-core Ivy-Bridge system
- Discuss three different GPU techniques and their tradeoffs
- Roofline analysis to gain insights into how the GPU kernel can be improved
- Discuss performance implications of different approaches and explanation through analyzing performance counter outputs

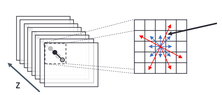
2. A Radar Satellite Image Processing Algorithm

Interferometric Synthetic Aperture Radar (InSAR)

- InSAR is an earth-orbiting satellite-based remote sensing technology used primarily for estimating displacement of the earth's surface [1]
- Applications – monitoring earth subsidence and uplift due to urban infrastructure development, mining, oil and gas extraction, and permafrost thawing



Every pixel stores a complex number



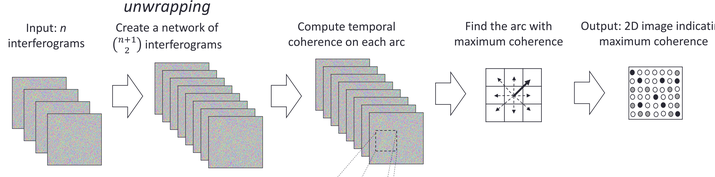
$arc = C\bar{N}$ (*C* is a pixel an *N* is its neighbour, *n* is the number of input interferograms)

Temporal arc coherence, $\tau = \left\| \frac{\sum_{k=1}^{\binom{n+1}{2}} C_k \overline{N_k}}{\binom{n+1}{2}} \right\|$

$\tau_{max} = \max_{0 \le k < F} (\tau_k)$ (*F* is neighbour count)

The Algorithm - Persistent Scatterer Pixel Selection

- Identify *permanent* objects, both natural and artificial, based on SAR imagery [2]
- Computes probability distribution of pixels – likelihood of a pixel representing a permanent object
- Classifying pixels improves efficiency and quality of later steps in the processing pipeline, e.g, *phase unwrapping*



A search window with eight arcs

Due to massive storage requirement of an Interferogram network (in the order of terabytes), its generation is combined with temporal coherence computation. Hence,

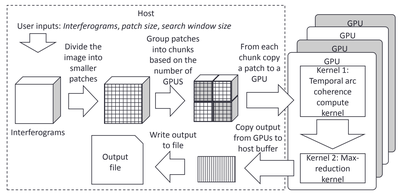
$\tau = \left\| \frac{\sum_{i=1}^{n} C_i \overline{N_i} + \sum_{i=1}^{n} \sum_{j=i+1}^{n} C_i \overline{N_i} C_j \overline{N_j}}{\binom{n+1}{2}} \right\|$

Typical workload size
- A 15,000 × 8,000-pixel interferogram – **1GB**
- Network of 100 interferograms – **5TB**

Compute-to-Memory ratio of the kernel that does temporal coherence computation is **1.5:1**
- Fairly balanced and difficult to optimize
- Should benefit from higher FLOP-rate and memory bandwidth

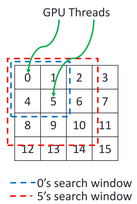
(Please refer to [3] for comprehensive description of this algorithm)

3. GPU Techniques



- Above is an overview of the GPU implementation
- We discuss three approaches for the main GPU kernel: Temporal arc coherence compute kernel

1. Window-parallel (WP)

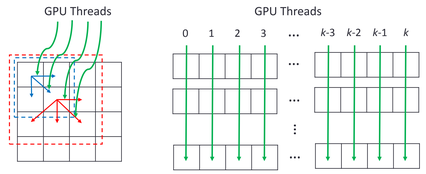


- Maps a pixel to a GPU thread
- Image data is stored in a 3D array
- Neighbors within a search-window are visited in a row-major fashion
- Each thread sequentially computes temporal coherences of the arcs within a search window

Limitations
- Number of threads can be small if spatial dimension of the image segment on the GPU is small due to having a very large network
- Some non-coalesced memory accesses

2. Arc-parallel (AP)

- Finer-granular parallelism - maps an arc to a GPU thread
- Computes temporal coherences of all the arcs in the image (a patch) in parallel
- Arcs and Image data are stored in structure of arrays (SOA) for memory coalescing
- Introduces redundancy in image data



SOA representation ensures 100% coalesced device-memory access

```
struct arcs {
  uint32_t* i;
  uint32_t* j;
  uint32_t* x;
  uint32_t* y;
  float* coherences;
};
Pixels (i,j) and (x,y) form an arc. 'coherences' is the container for storing corresponding computed τ
```

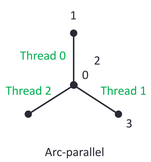
```
struct image {
  float* ij_real;
  float* ij_img;
  float* xy_real;
  float* xy_img;
};
Each pixel is a complex number. 'real' and 'img' represent the real and imaginary parts of a pixel respectively
```

Input interferograms are stored in sequence

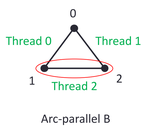
Limitations
- Although improves memory coalescing, redundant data results in increased traffic to device memory by 2x
- Does not offer speedup over window-parallel

3. Arc-parallel B (AP-B)

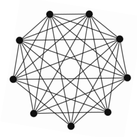
- Aims at reducing memory traffic to the slower device memory and improving L2 caching on the GPU
- Image data is stored in a 3D array as in window-parallel - eliminates redundancy as in the original arc-parallel
- Arcs are grouped and mapped to threads in a manner that should improve locality (i.e., on-die data caching) within a warp



Arc-parallel



Arc-parallel B

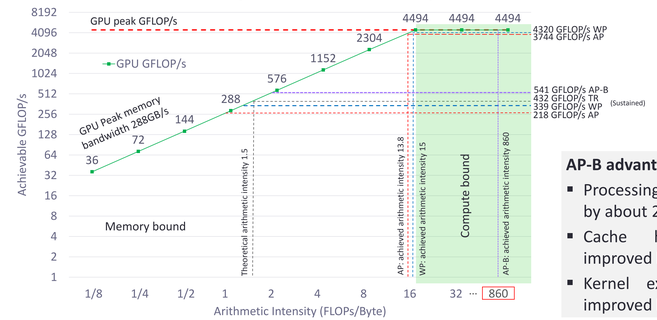


Arcs are grouped so that they form a graph (K_9 complete graph minus four edges) consisting of 32 edges. These 32 arcs are mapped to 32 consecutive threads in a warp. The objective is to reduce stalls at the warp level due to data requests to the device memory through improving cache hit rate.

In the new approach, it is highly likely that Thread 2, mapped to the arc (1,2), can read all the data, already requested by the other two threads, from the nearest, fast on-die memory

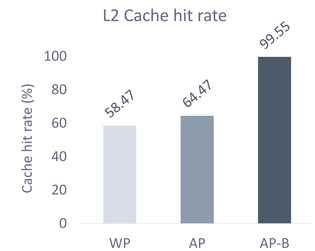
4. Evaluation and Discussions

Roofline analysis of the GPU techniques



- Arithmetic intensity for an application is the ratio of its generated FLOP-count to bytes read from the DRAM
- The roofline plot [4] indicates the implementation can be benefited from improved locality, i.e., better caching which AP-B focuses on improving

AP-B advantages over WP
- Processing rate improved by about 200GFLOP/s
- Cache hit rate is improved by about 41%
- Kernel execution time improved by 2.4x



Graph	Eligible Warps per Cycle	SM Efficiency	Stall Memory Dependency	DRAM Read Throughput
Window-parallel	4	87%	15%	23GB/s
Arc-parallel B	6	99%	11%	635MB/s

- AP-B improves parallel-efficiency as it utilizes more GPU threads
- AP-B reduces memory traffic to the slower off-chip device memory
- AP-B reduces memory requests related stalls which in turn improves warp scheduling

Future work

- Utilize shared memory to further improve locality and eliminate effects of strided accesses
- Extend the AP-B thread mapping approach from the warp level to the block or SMX level on the GPU

References

[1] P. Rosen, S. Hensley, I. Joughin, F. K. Li, S. Madsen, E. Rodriguez, and R. M. Goldstein, "Synthetic aperture radar interferometry," Proc. of the IEEE vol. 88, no. 3, pp. 333-382, March 2000.

[2] A. Hooper, "Persistent scatterer radar interferometry for crustal deformation studies and modeling of volcanic deformation," Ph.D. Thesis, May 2006, Retrieved from http://web.stanford.edu/group/radar/people/Hooper_Thesis.pdf, on 16 December 2014.

[3] T. Reza, A. Zimmer, P. Ghuman, and M. Ripeanu, "Accelerating Persistent Scatterer Pixel Selection for InSAR Processing," In Proc. of the 26th IEEE International Conference on Application-specific Systems, Architectures and Processors, Toronto, Ontario, 27 - 29 July 2015.

[4] S. Williams, A. Waterman, and D. Patterson. "Roofline: an insightful visual performance model for multicore architectures". Commun. ACM, vol. 52, pp. 65–76, 2009.