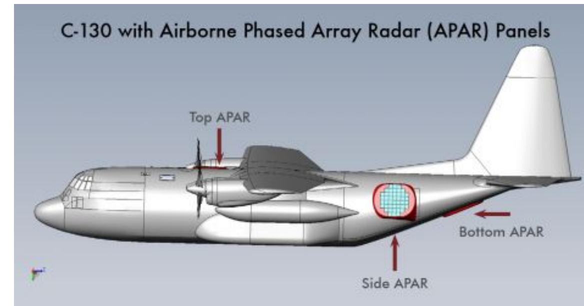
Transforming an observational assimilation application on CPU and GPU

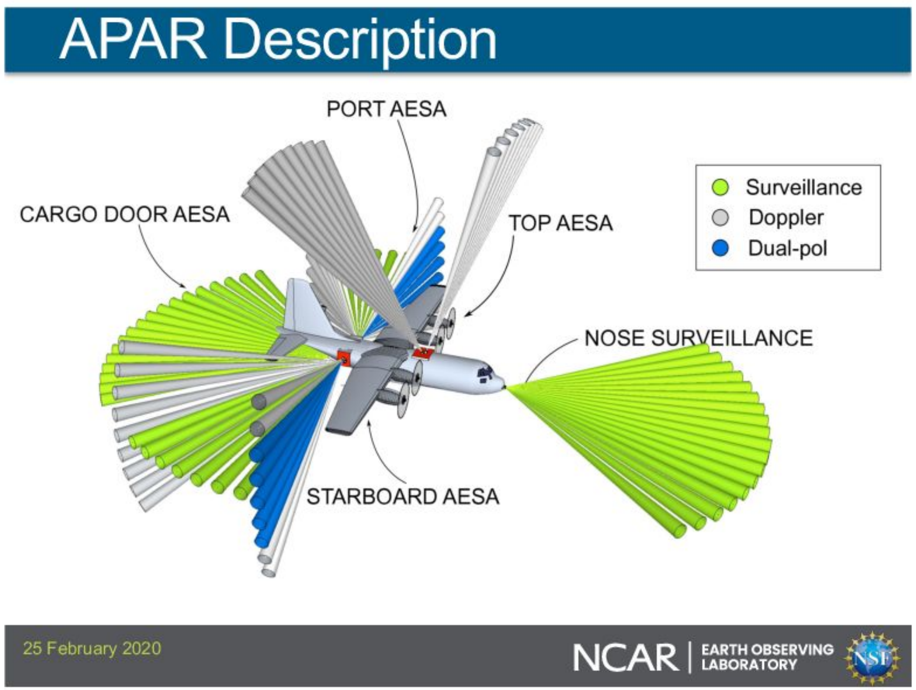
John Dennis, Brian Dobbins, Allison Baker, Youngsung Kim, Jian Sun

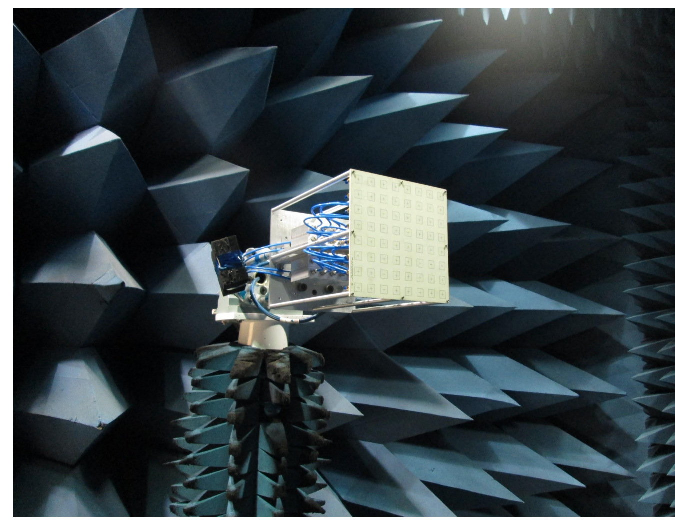
June 1, 2021
NHUG meeting

Airborne Phased Array Radar (APAR)

- Airborne precipitation radar to replace retired ELDORA aircraft
- Science drivers
  - Hurricanes and tropical cyclones
  - Continental convection
  - Extreme precipitation events
  - Arctic studies
  - Cloud, aerosol, and radiation studies



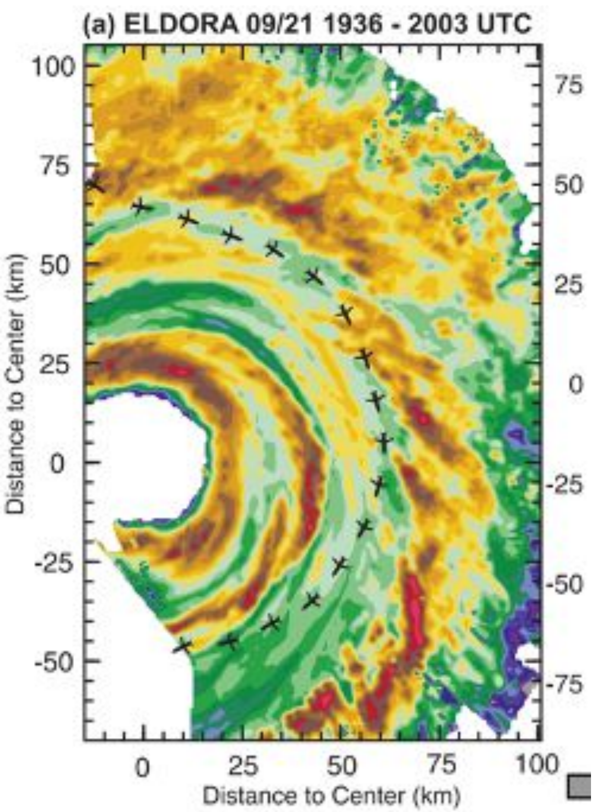




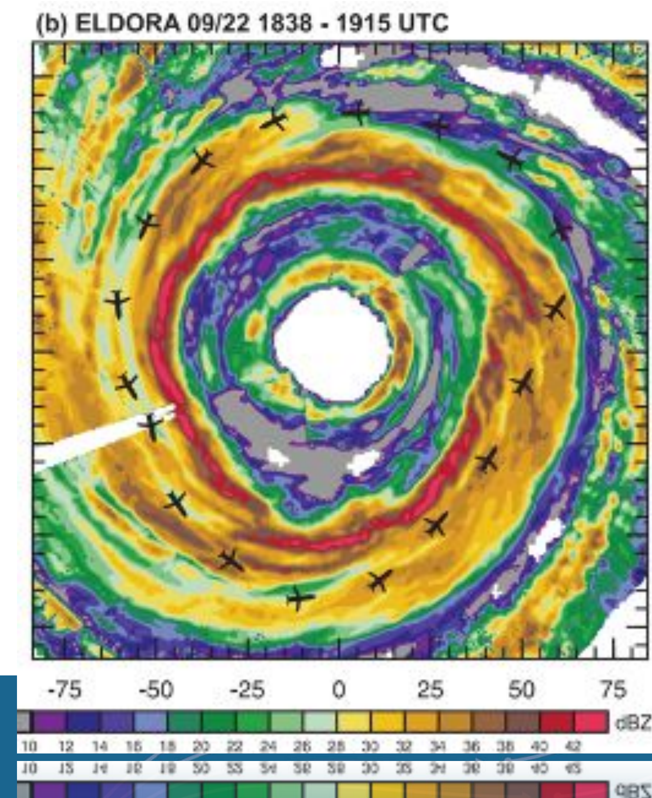
Airborne Phased Array Radar (APAR), Wen-Chau Lee, Vanda Grubisic, Lou Lussier, https://www.ofcm.gov/meetings/TCORF/ihc20/session_3/3-7_lee.pdf

Spline Analysis at Mesoscale Utilizing Radar and Aircraft Instrumentation (SAMURAI)

- Developed by M. Bell @ CSU
- Consumes Airborne observations doppler
- Generates variational analysis of seven variables [wind, precipitation, vorticity, etc]
- Variational analysis product can be used by NWP
- C++, OpenMP based parallelism



Bell et. al 2012



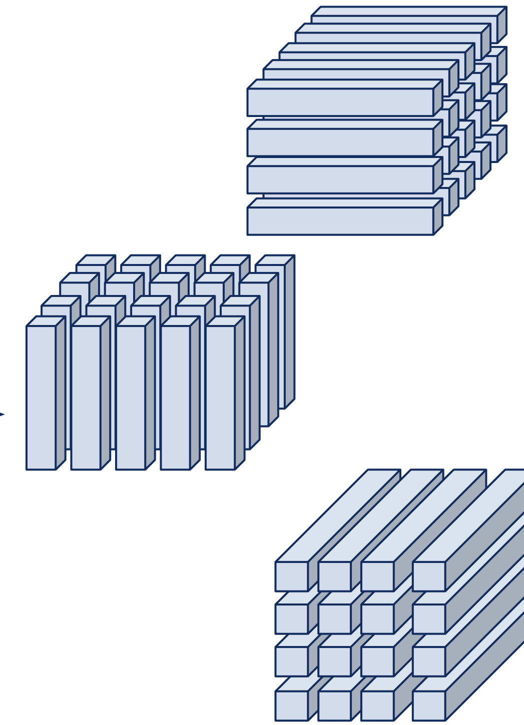
SAMURAI optimization effort

- Funded by Earth Observing Laboratory (EOL) through a NOAA grant
- Original version of SAMURAI takes 2-3 days to perform analysis (single node) for test datasets
- Anticipated APAR generate data ~16x larger
- What can be done to accelerate the processing of observations?
- Is this application suitable for GPUs?

Goal: Run analysis in less than 6 hours

SAMURAI computational characteristics

- Matrix-free solver implemented by several operators
- Main data-structures
  - 3D physical grid (eg: 241x241x33)
  - Observation matrix H [can be quite large]
- Computational routines
  - NCG or Truncated-Newton solver
  - Pencil calculations on physical grid
    - SAtransform
    - SCtransform
  - Multiply by H: Htransform
  - Multiply by H^T: calcHTransform



SAMURAI performance issues

- Inefficient indexing and limited thread parallelism over physical grid
  - SAtransform
  - SCtransform
- Limited thread parallelism over H^T operator
  - calcHTranpose
- Non-unit stride for observation vector
  - Htransform
  - calcHTranpose
- Numerical inefficient Nonlinear Conjugate Gradient solver
- No threading within existing solver

Numerical solver

Big Picture:

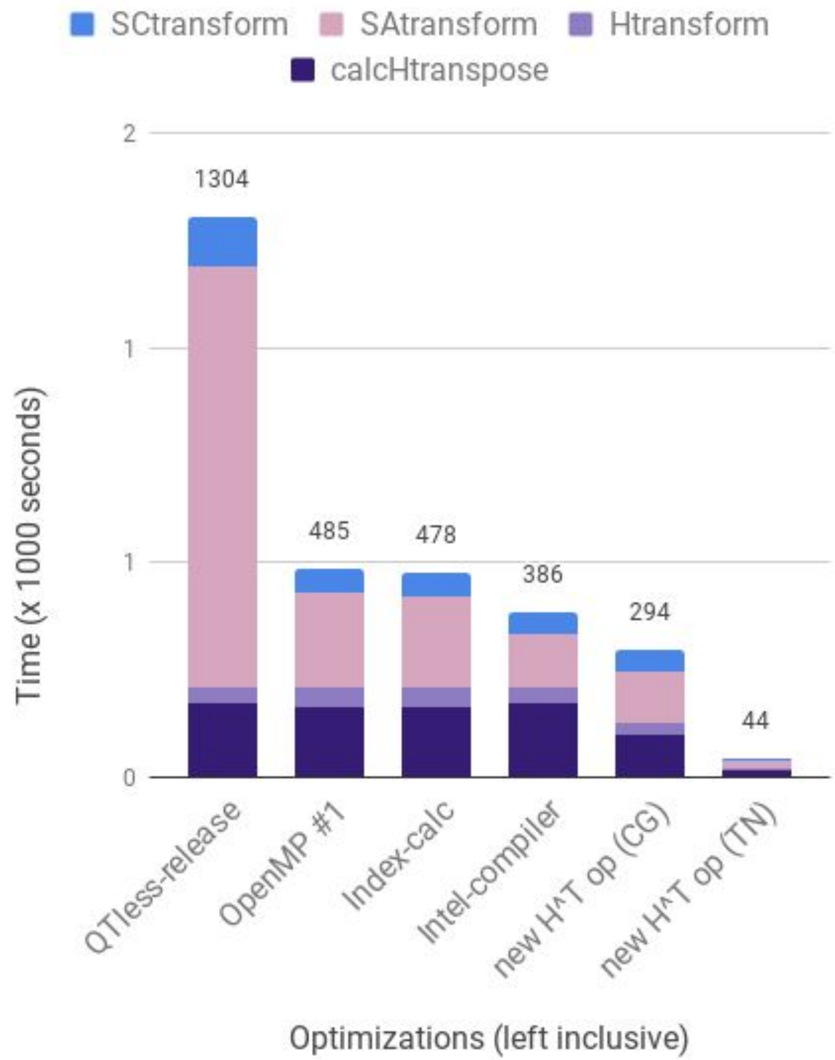
- minimize cost (objective) function: $J(\mathbf{x})$
- by solving for gradient: $\nabla J(\mathbf{x}) = 0$
- nonlinear optimization: *at each iteration, "step" closer to the solution in a chosen "search direction" (iterative process)*

New solver: truncated Newton Method (TN)

- "step" closer to the solution in a chosen search direction (iteratively)
- Newton direction (d): $\nabla^2 J(\mathbf{x}_k)\, d_{k+1} = -\nabla J(\mathbf{x}_k)$
  - solve iteratively with Conjugate Gradient
  - we don't form $\nabla^2 J(\mathbf{x}_k)$ - just the matvec product
- step length in direction (d) determined by line search
  - linesearch = Moré-Thuente
- look at relative reduction in the gradient (more standard):

$$||\nabla J(\mathbf{x}))|| \,/\, |\nabla J(\mathbf{x}_0))|| < 1e\text{-}4$$

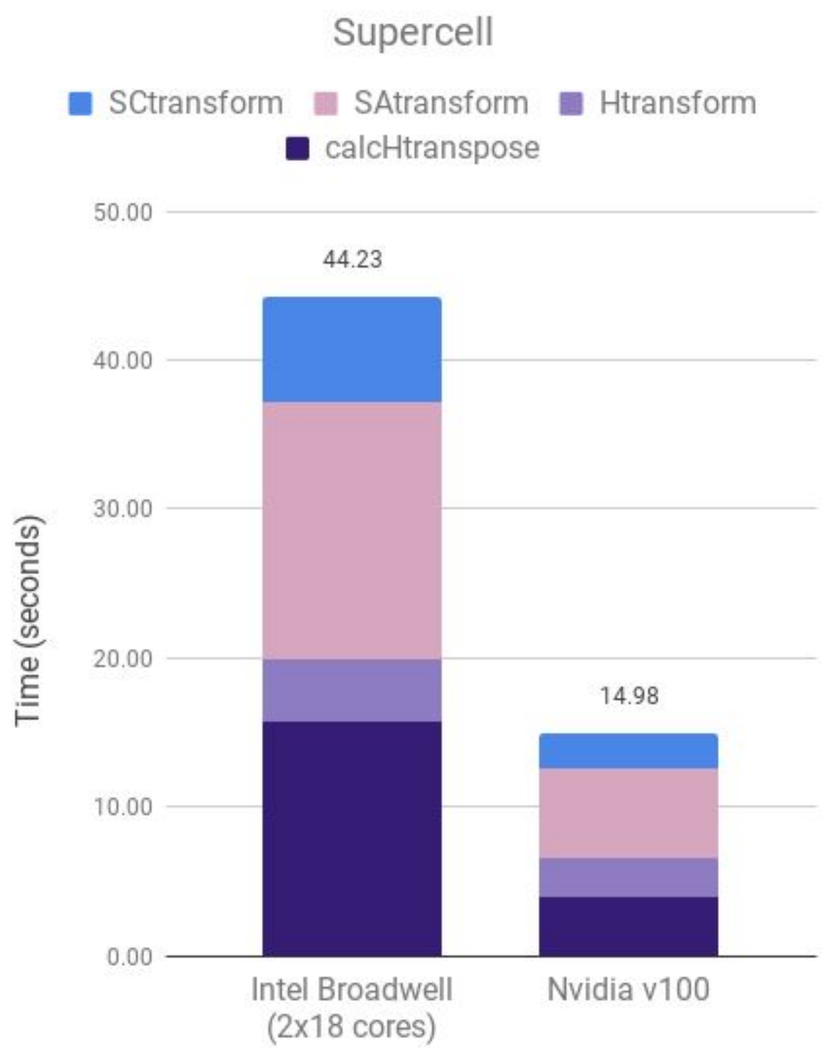
Cost breakdown: Supercell (20 iterations)

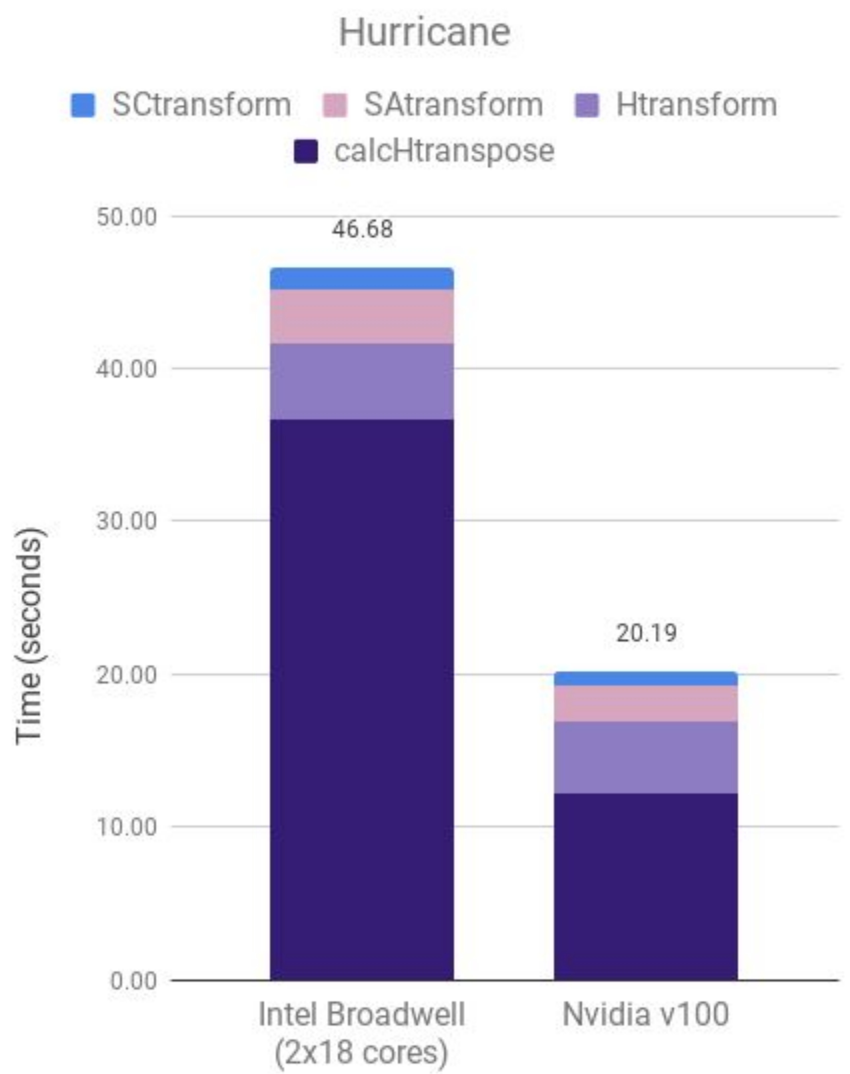


GPU enablement

- Utilized OpenACC parallel and data movement directives
- Very large working set size makes application ideal for GPU execution
- All computationally demanding calculations are GPU resident
- Currently using managed memory
- Very small amount host → device memory transfers still exist
- Non-trivial rewrite of the calcHTranspose was necessary

Cost breakdown: [new H^T op] (20 iterations)





Summary of SAMURAI code optimizations

Code version	Platform	Execution time (minutes)	
		Supercell	Hurricane
Original	Intel Broadwell (2x18)	577	609
Serial + OpenMP opt	Intel Broadwell (2x18)	151	382
TN solver	Intel Broadwell (2x18)	46	74
new H^T op	Intel Broadwell (2x18)	19	20
	NVIDIA v100	5.4	4.9
Overall speedup (original CPU/ final GPU)		106	124

Conclusions

- Modernized and portable version of SAMURAI created
- Significant (106 - 124x) speedup achieved on SAMURAI
- Team with diverse and complementary skills can have profound impact on application performance
- Possible to use full resolution of APAR instrument with CPU or GPU based HPC resource
- HPC resources are no-longer needed for modest resolution configurations
- Funding: NOAA grant through EOL

Team members

- Allison Baker (NCAR)
- Brian Dobbins (NCAR)
- Youngsung Kim (ORNL)
- Jian Sun (NCAR)

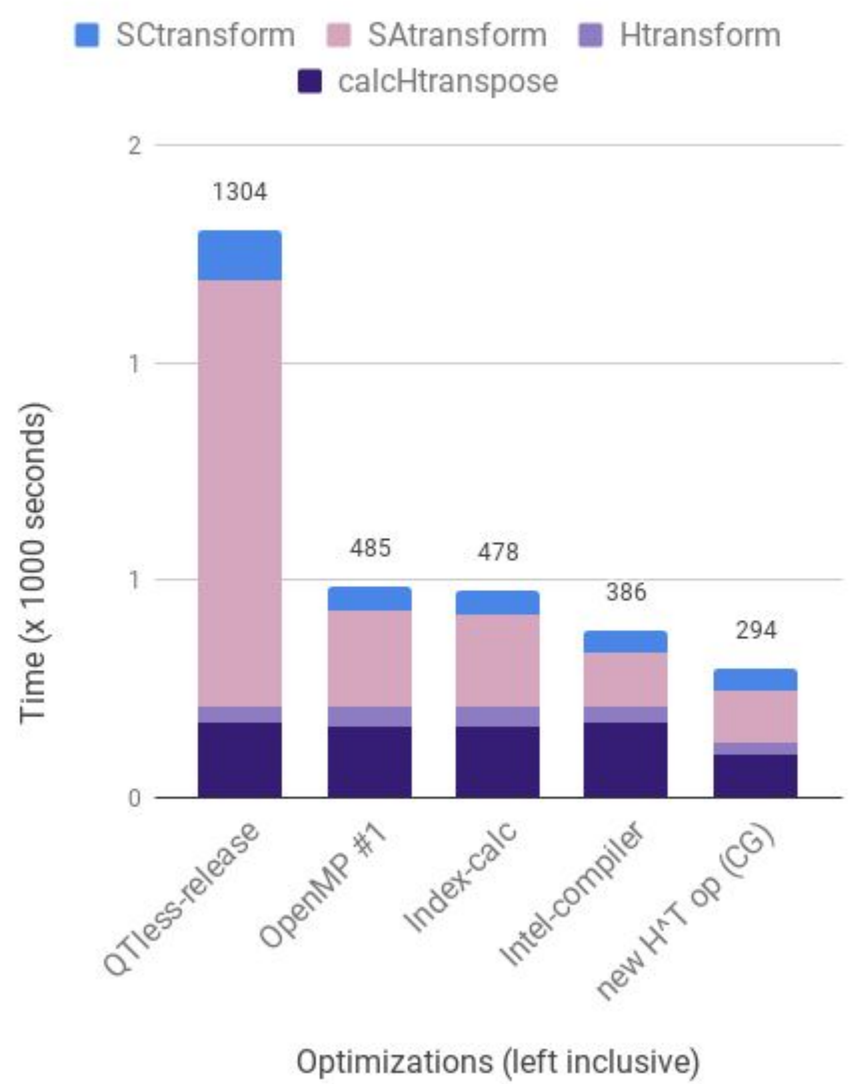
Collaborators

- Wen-chau Lee, APAR PI (NCAR)
- Scott Ellis (NCAR)
- Michael Bell (CSU)
- Ting-yu Cha (CSU)
- Alex DesRosiers (CSU)
- Michael Dixon (NCAR)

Questions?

John Dennis (dennis@ucar.edu)

Cost breakdown: Supercell (20 iterations)



Acknowledgements

- Funding: NOAA grant through EOL
- Team members
  - Allison Baker (NCAR)
  - Brian Dobbins (NCAR)
  - Youngsung Kim (ORNL) formally NCAR
- Others
  - Wen-chau Lee, APAR PI (NCAR)
  - Scott Ellis (NCAR)
  - Michael Bell (CSU)
  - Ting-yu Cha (CSU)
  - Alex DesRosiers (CSU)
  - Michael Dixon (NCAR)
  - Jian Sun (NCAR)

Conclusions

- Modernized and portable version of SAMURAI created
- Significant (106 - 124x) speedup achieved on SAMURAI
- Team with diverse and complementary skills can have profound impact on application performance
- Possible to use full resolution of APAR instrument with CPU or GPU based HPC resource
- HPC resources are no-longer needed for modest resolution configurations
- GPU enablement allows SAMURAI to be used in-flight
  - Potential to adjust science goals in real-time

Experimental configuration

- Computational platforms
  - 2 x 2.3-GHz Intel Xeon E5-2697V4 (Broadwell) processors
  - 1 NVIDIA v100 32 GB
- Compilers:
  - PGI 20.4
  - Intel 19.0.5
- SAMURAI datasets
  - Supercell:
    - physical grid: 241x241x33
    - # observations: 4.3M
  - Hurricane
    - physical grid: 105x201x33
    - # observations: 8.7M

Future work

- Improve efficiency of GPU implementation
  - Rewrite calcHTranpose again
  - Eliminate all excessive PCIe traffic
  - Improve memory access patterns on GPU
- I/O is serial and now a significant % of total execution time (40%)
- High resolution APAR data sets have very large memory requirements
  - Need multi-node CPU implementation
  - Need multi-GPU implementation

Limited thread parallelism over physical grid

parallelized over varDim

```
#pragma omp parallel for
for (int var = 0; var < varDim; var++) {
  …  temporary array allocations;
 for (int iIndex = 0; iIndex < iDim; iIndex++) {
 … } temporary array allocations;
 for (int kIndex = 0; kIndex < kDim; kIndex++) {
 … } temporary array allocations;
 for (int jIndex = 0; jIndex < jDim; jIndex++) {
 … }
}
```

parallelized over i, j, k grid dims

```
for (int var = 0; var < varDim; var++) {
  …
#pragma omp parallel for
  for (int iIndex = 0; iIndex < iDim; iIndex++) {
  temporary array allocations; … }
#pragma omp parallel for
  for (int kIndex = 0; kIndex < kDim; kIndex++) {
  temporary array allocations; … }
#pragma omp parallel for
  for (int jIndex = 0; jIndex < jDim; jIndex++) {
  temporary array allocations; … }
}
```

Issues with the H^T operator: calcHtranpose

Original:

Cons: not threaded, indirect address for store, non-unit access stride to obsVector

```
for(int m=0; m<mObs; ++m) {
  int mi = m*(7+varDim*derivDim)+1;
  const int begin = IH[m];
  const int end = IH[m + 1];
  for(int j=begin; j<end; ++j) {
    //#pragma omp atomic
    Astate[JH[j]] += H[j] * yhat[m] *
                            obsVector[mi];
  }
}
```

Second attempt:

Pros: partially threaded

Cons: indirect address for store, non-unit access stride to obsVector, would generate PCIe traffic for GPU

```
#pragma omp parallel for
 for(int m=0; m<mObs; ++m) {
  real val = yhat[m] *
          obsVector[m*(7+varDim*derivDim)+1];
  for(int j=IH[m]; j<IH[m+1]; ++j) {
   tempHval[j] = H[j] * val;
  }
 }
 for(int i=0; i<IH[mObs]; ++i) {
   Astate[JH[i]] += tempHval[i];
 }
```

Issues with the H^T operator: calcHtranpose (con't)

```
#pragma omp parallel for
#pragma acc parallel loop gang vector
 for(int n=0;n<nState;n++){
    int ms = mPtr[n];
    int me = mPtr[n+1];
    real tmp = 0;
    if(me>ms){
        for (int k=ms;k<me;k++){
           int m=mVal[k];
           int j=I2H[k];
           real val = yhat[m] * obsData[m];
           tmp += H[j] * val;
        }
     }
     Astate[n]=tmp;
  }
```

Third attempt:

Pros: Fully threaded, eliminated indirect address for store, unit access stride for obsData, GPU device resident.

Cons: suboptimal memory data access patterns, uses a lot of memory for address arrays mPtr,mVal

Future activity, explicitly store H^T and do a standard CSR format

Numerical solver

Big Picture:

- minimize cost (objective) function: $J(\mathbf{x})$
- by solving for gradient: $\nabla J(\mathbf{x}) = 0$
- nonlinear optimization: *at each iteration, "step" closer to the solution in a chosen "search direction" (iterative process)*

Old Samurai Solver:

- nonlinear Conjugate Gradient (NCG)
  - compute search direction (multiple options)
  - determine optimal step length

    line search = Brent's Method (expensive)
- convergence criteria:
  - ~change in cost function between consecutive NCG steps < 1e-5
  - harder to do a comparisons across as the reduction in the gradient is not going to be the same in every case (problems/solvers)

Numerical solver (con't)

New solver: truncated Newton Method (TN)

- "step" closer to the solution in a chosen search direction (iteratively)
- Newton direction (d): $\nabla^2 J(\mathbf{x}_k)\ d_{k+1} = -\nabla J(\mathbf{x}_k)$
  - solve iteratively with Conjugate Gradient
  - we don't form $\nabla^2 J(\mathbf{x}_k)$ - just the matvec product
- step length in direction (d) determined by line search
  - linesearch = Moré-Thuente
- look at relative reduction in the gradient (more standard):

$$||\nabla J(\mathbf{x}))|| \ / \ |\nabla J(\mathbf{x}_0))|| < 1e\text{-}4$$

NCAR | UCAR

Outline

- Background/Motivation
  - APAR
  - SAMURAI
- Initial state of code
- Code optimization
- Numerical solver
- GPU enablement
- Conclusions

Outline

- Background/Motivation
  - APAR
  - SAMURAI
- Initial state of code
- Code optimization
- Numerical solver
- GPU enablement
- Conclusions

Outline

- Background/Motivation
  - APAR
  - SAMURAI
- Initial state of code
- Code optimization
- Numerical solver
- GPU enablement
- Conclusions

Outline

- Background/Motivation
  - APAR
  - SAMURAI
- Initial state of code
- Code optimization
- Numerical solver
- GPU enablement
- Conclusions

Challenges

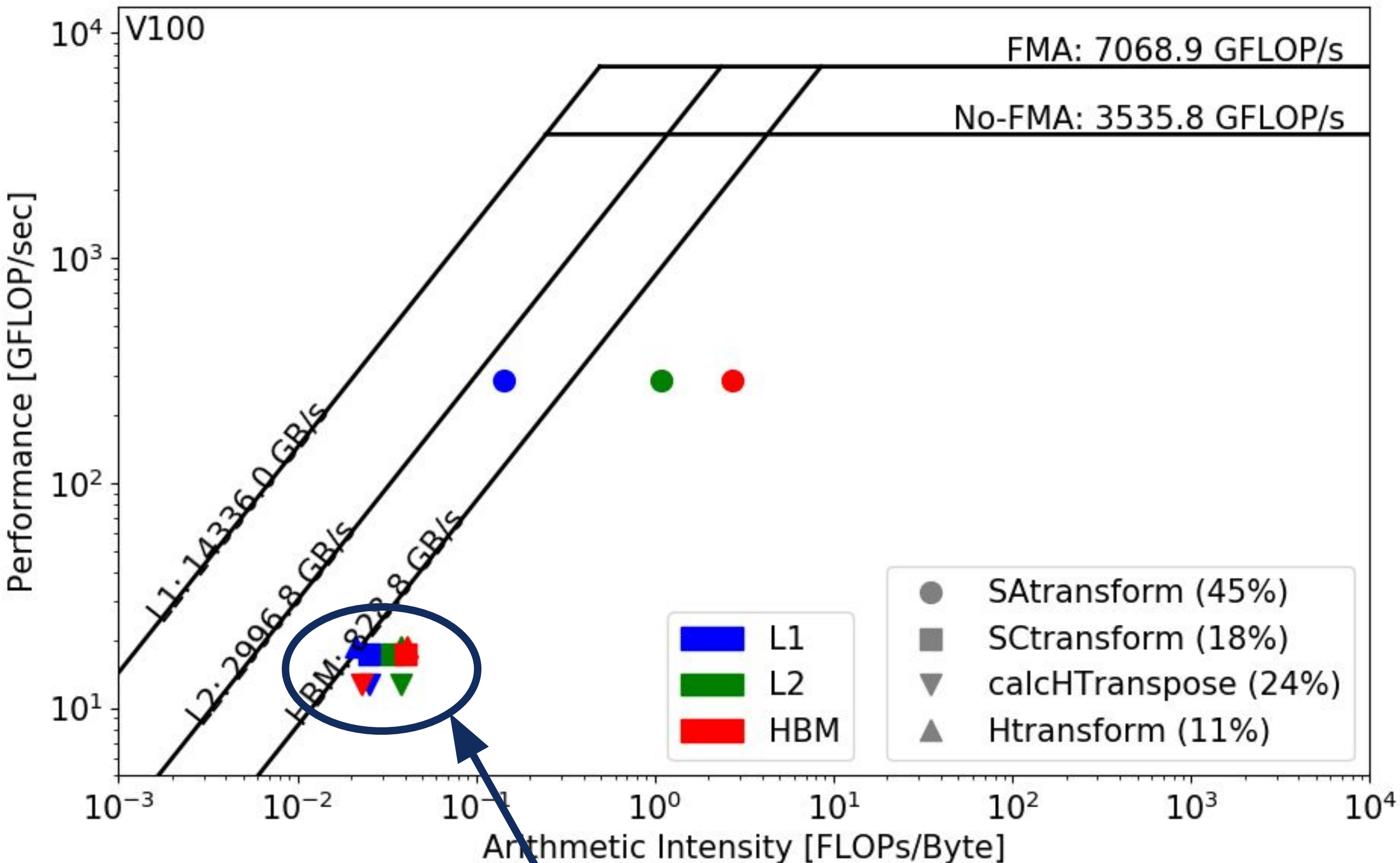
- Require a GUI C++ library for string manipulation (?)
- Execution only possible in a Docker container
  - Not possible to use NCAR supercomputer environment due to security restrictions
  - performance analysis tools and SAMURAI incompatible
- Very long runtime: 2-3 days
  - prevented execution in NCAR queueing system
  - only possible to run on laptop or cloud provider
- Larger problems exceed memory of typical laptop

Porting of SAMURAI to HPC cluster

- Original version of code was a binary executable in a Docker container→ made development nearly impossible
- Removing Qt library dependency → C++11
- Redesign Cmake build structure
  - Support use of multiple compilers
  - Support use of standard performance analysis tools
- Significant effort: ~2 months

Now possible for for multiple team members to contribute to project!

How efficient is GPU implementation? Roofline (supercell)



Lots of time spent running at HBM rates and/or PCIe traffic

Plan of attack

- Port Docker container version of SAMURAI to Charliecloud
- Port Charliecloud version to standard HPC cluster (Cheyenne)
- Analyze performance of SAMURAI
- Optimize code
- Evaluate replacing existing Conjugate Gradient solver
- Evaluate use of GPU using OpenACC

Solver on a variety of problems

Timing results (Cheyenne)

Problem	sizes	Samurai NCG cost*	Samurai NCG rel. norm**	Truncated Newton cost*	Truncated Newton rel. norm**	Speedup
Supercell	(241x241x33) obs = 4372390	2.3 h	4.3e-4	23.4m	9.7e-5	5.9x
Supercell: 2x	(481x481x65) obs = 17494182	13.4 h	1.3e-3	2.8 h	9.9e-5	4.8x
Hurricane	(105x201x33) obs = 8675128	6.6 h	4.5e-5	26.9 m	9.6e-5	14.7x
Hurricane: 2x	(211x401x65) obs = 13471745	11.5 h	8.0e-5	1.3 h	9.6e-5	8.8x

*Cost = 3D minimize() time **rel. norm = value of $\|\nabla J(\mathbf{x})\| / \|\nabla^2 J(\mathbf{x}_0)\|$ at convergence