


Enabling efficient execution of a variational data assimilation application

John M Dennis¹ , Allison H Baker¹ , Brian Dobbins¹ ,
Michael M Bell², Jian Sun¹ , Youngsung Kim³ and Ting-Yu Cha² 

The International Journal of High Performance Computing Applications
2022, Vol. 0(0) 1–14
© The Author(s) 2022
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/10943420221119801
journals.sagepub.com/home/hpc


Abstract

Remote sensing observational instruments are critical for better understanding and predicting severe weather. Observational data from such instruments, such as Doppler radar data, for example, are often processed for assimilation into numerical weather prediction models. As such instruments become more sophisticated, the amount of data to be processed grows and requires efficient variational analysis tools. Here we examine the code that implements the popular SAMURAI (Spline Analysis at Mesoscale Utilizing Radar and Aircraft Instrumentation) technique for estimating the atmospheric state for a given set of observations. We employ a number of techniques to significantly improve the code's performance, including porting it to run on standard HPC clusters, analyzing and optimizing its single-node performance, implementing a more efficient nonlinear optimization method, and enabling the use of GPUs via OpenACC. Our efforts thus far have yielded more than 100x improvement over the original code on large test problems of interest to the community.

Keywords

data assimilation, GPU, optimization

1. Introduction

The Airborne Phased Array Radar (APAR) is under development by the Earth Observing Laboratory at the National Center for Atmospheric Research (NCAR). The APAR panels will be mounted on aircraft to obtain observations from severe weather and storms over areas that are difficult to detect with ground-based radars (mountains, oceans, and arctic regions etc.) The APAR panels will enable multi-Doppler synthesis of the three-dimensional wind field at finer spatial resolution than currently available. Research with these wind fields will improve our understanding of storm dynamics and severe weather and ultimately enable its assimilation into high-resolution numerical weather prediction (NWP) models to improve weather forecasts (Vivekanandan and Loew 2018).

The process of synthesizing airborne radar data into a gridded wind field involves utilizing the geometry of multiple Doppler beams, each containing only a partial projection of the full wind field (Jorgensen et al., 1996). Our focus in this work is on the three-dimensional variational analysis technique called Spline Analysis at Mesoscale Utilizing Radar and Aircraft Instrumentation, or the SAMURAI technique (Bell et al., 2012; Foerster et al., 2014; Foerster and Bell 2017). For a given set of

observations and error estimates, the SAMURAI technique minimizes a variational cost function to provide a maximum likelihood estimate of the atmospheric state. The SAMURAI technique has been used extensively to synthesize dual-Doppler data collected from the ELDORA (Electra Doppler Radar), which was mounted on the NCAR Electra and Naval Research Laboratory P-3 aircraft during its lifetime. Similar airborne radar configurations are currently available on the National Oceanic and Atmospheric Administration (NOAA) P-3 aircraft. An example of the processed wind field from Hurricane Rita (2005) analyzed in Boehm and Bell (2021) is shown in Figure 1. We refer to the code that implements the SAMURAI technique as SAMURAI, which is a publicly available C++ code* with OpenMP-based single-node

¹National Center for Atmospheric Research, Boulder, CO, USA

²Colorado State University, Fort Collins, CO, USA

³Oak Ridge National Laboratory, Oak Ridge, TN, USA

Corresponding author:

John M. Dennis, National Center for Atmospheric Research, Computational and Information Systems Laboratory, Boulder, CO 80303, USA.

Email: dennis@ucar.edu

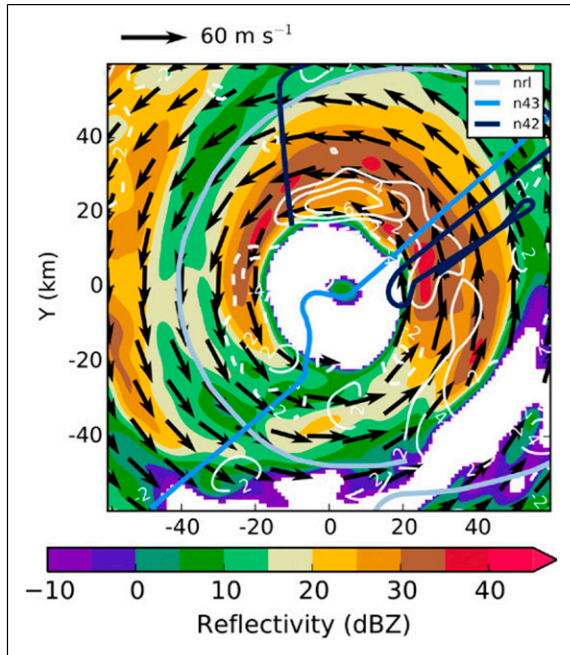


Figure 1. Horizontal cross section of SAMURAI analysis of Hurricane Rita (2005), adapted from Boehm and Bell (2021), showing radar reflectivity (shaded, dBZ), horizontal velocity (black arrows, m s^{-1}), and vertical velocity (white contours, m s^{-1}) at 5 km altitude at 2040 UTC on 23 September 2005. Flight tracks from NRL and NOAA 42 and 43 aircraft with airborne radars are overlaid.

parallelism, originally developed by M. Bell. Although SAMURAI performance was reasonable at the time of its development, for example, for data collected from ELDORA, it is anticipated that it will be unacceptably slow for processing observational data from APAR. For example, using test configurations that are significantly larger than the ELDORA instrument can take 2–3 days on an older university deployed system. The eventual operational APAR data volume will be as much as 16x the current test configurations due to the high-resolution observations. Therefore, the focus of our work here is on improving the performance and efficiency of the code such that it is ready to handle larger data volumes. In particular, the scientists using SAMURAI to synthesize the observational data requested that analysis on APAR-sized test data should be completed in less than 12 compute hours.

Our approach to accelerating the processing of such observations and generally improving the overall performance of the code was threefold. First, we refactored SAMURAI to more easily run on a typical HPC cluster, including the NCAR supercomputing environment. Enabling this capability was important for facilitating the use of standard performance tools and accommodating larger memory requirements, both of which allowed us to analyze code performance and find standard ways to optimize the

code. Second, SAMURAI was minimizing the cost function via the nonlinear Conjugate Gradient (NCG) method (e.g., Nocedal and Wright (2006)). We re-examined this solver choice and proposed a more efficient solver for nonlinear optimization. Our third major effort was to enable the use of Graphics Processing Units (GPUs) via OpenACC directives, providing even further performance gains. The application of the standard code optimization reduced execution time by a factor of 5.3x, the numerical changes by a factor of 6.4x, and GPU enablement by a factor of 3.1x. Thus far, the combined impact of the described three-pronged approach has led to speedups of more than 100x over the original SAMURAI code. Although we have already far exceeded the targeted 12-hour time limit requested by the scientists for the analysis of large datasets, we plan to pursue further improvements, including multi-node capabilities.

In this paper, we describe our efforts to significantly improve the performance of SAMURAI. We first provide background information about SAMURAI in Section 2 and describe our testing platform and experiments in Section 3. Then, in Section 4, we detail specific code optimizations that improve the performance, and we discuss the numerical optimization solver in Section 5. In Section 6, we describe porting the code to GPU and the resulting speedup. We conclude (Section 7) by summarizing our work and discussing future plans.

2. SAMURAI

As noted in the previous section, SAMURAI implements a variational analysis procedure originally described in Bell et al. (2012). The product of this technique is an atmospheric state composed of seven variables (wind velocity in both horizontal and vertical directions, temperature, water vapor mixing ratio, air density, and reflectivity) that can be analyzed for atmospheric research or assimilated into a NWP model to improve the simulation of an extreme event, for example. The atmospheric state is a maximum likelihood estimate that minimizes a variational cost function for a given set of observations (e.g., from APAR or ELDORA) and error estimates. SAMURAI has a number of advantageous features, including observational error specifications for different instrumentation, complex observation operators for remote-sensing, enforcement of physical constraints (such as mass continuity), and optional inclusion of a priori estimates of the atmospheric state that can be integrated with observations in a Bayesian framework.

Furthermore, we note that SAMURAI has the ability to use either an axisymmetric cylindrical coordinate system or a 3D Cartesian coordinate system and uses a finite element approach (which takes a set of overlapping cubic B-splines as a basis (Ooyama 2002) and transforms to and from the spline coefficients and physical space at each minimization

Table 1. Relating the operators in the cost function (1) to the functions in SAMURAI.

Operator	Corresponding function(s)
$\mathbf{v}_{\text{out}} = \mathbf{D}\mathbf{F}\mathbf{v}_{\text{in}}$	$\mathbf{v}_{\text{out}} = \text{SCtransform}(\mathbf{v}_{\text{in}})$
$\mathbf{v}_{\text{out}} = \mathbf{S}\mathbf{v}_{\text{in}}$	$\mathbf{v}_{\text{out}} = \text{SAttransform}(\mathbf{v}_{\text{in}})$
$\mathbf{v}_{\text{out}} = \mathbf{H}\mathbf{v}_{\text{in}}$	$\mathbf{v}_{\text{out}} = \text{Htransform}(\mathbf{v}_{\text{in}})$
$\mathbf{v}_{\text{out}} = \mathbf{H}^T\mathbf{R}^{-1}\mathbf{v}_{\text{in}}$	$\mathbf{v}_{\text{out}} = \text{calcHTranspose}(\mathbf{v}_{\text{in}})$

step). More technical details can be found in [Bell et al. \(2012\)](#), [Foerster et al. \(2014\)](#) and [Foerster and Bell \(2017\)](#).

Cost function

For our work here, it is important to understand that SAMURAI minimizes the single cost function $J(\hat{x})$ for control variable \hat{x} that represents the unknown coefficients of a finite element representation of the atmospheric state. The use of a control variable allows for an incremental form of a Bayesian cost function that provides the optimal additive increment to a prior first guess of the atmospheric state (called the “background state”) that can be obtained from any available source, such as a numerical weather model or in situ observations. The cost function then accounts for both the error in the observations (at the observation locations) and the error in the prior knowledge of the first guess of the atmospheric state.

This cost function can be written as

$$J(\hat{x}) = \frac{1}{2}\hat{x}^T\hat{x} + \frac{1}{2}(\mathbf{H}\mathbf{C}\hat{x} - \mathbf{d})^T\mathbf{R}^{-1}(\mathbf{H}\mathbf{C}\hat{x} - \mathbf{d}), \quad (1)$$

where \mathbf{H} is the linearized observation operator, \mathbf{C} is the square root of the background error covariance matrix, \mathbf{R} is the observational error covariance matrix, and \mathbf{d} is the difference between the observations (\mathbf{y}) and the nonlinear observation operator applied to the background state estimate ($h(\hat{x}_b)$) such that $\mathbf{d} \equiv \mathbf{y} - h(\hat{x}_b)$. The goal is to determine the value for state \hat{x} that minimizes cost function $J(\hat{x})$, which means that we want the gradient of the cost function to be zero. Note that the gradient, $\nabla J(\hat{x})$, can be written as

$$\nabla J(\hat{x}) = (\mathbf{I} - \mathbf{C}^T\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H}\mathbf{C})\hat{x} - \mathbf{C}^T\mathbf{H}^T\mathbf{R}^{-1}\mathbf{d}, \quad (2)$$

where \mathbf{I} indicates the identity matrix.

Many of the operators in the cost function in SAMURAI are implemented as matrix-free, as most would be too large or expensive to compute explicitly ([Gao et al., 2004](#)). The incremental form of the cost function avoids inversion of the extremely large background error covariance matrix and allows for the linear operators to be expressed as a sequence of operations. For example, operator \mathbf{C} cannot be stored for

practical data assimilation applications and is instead determined by the sequence of operations

$$\mathbf{C}\hat{x} = \mathbf{S}\mathbf{D}\mathbf{F}\hat{x}. \quad (3)$$

Operator \mathbf{S} indicates the spline transform, \mathbf{D} represents a diagonal matrix containing the standard deviations of the background error, and \mathbf{F} indicates the recursive spatial filter operations that directly affect the influence of the observations. Also, we note that the observation matrix, \mathbf{H} , can be quite large due to the voluminous amounts of radar data (e.g., the number of non-zeros anticipated for APAR is $\mathcal{O}(10^9)$).

Therefore, multiplication by the \mathbf{H} and \mathbf{H}^T operators is quite costly. See [Gao et al. \(2004\)](#) and [Bell et al. \(2012\)](#) for more specific details on the operators.

Relating the code to the operators

In the remainder of the paper, we will discuss the following SAMURAI functions (indicated by italics) that implement the operations needed to minimize the cost function: *SCtransform*, *SAttransform*, *Htransform*, and *calcHTranspose*. Consider two vectors \mathbf{v}_{in} and \mathbf{v}_{out} , indicating an input vector and output vector, respectively, for a function. [Table 1](#) clarifies the purpose of each function in relation to the operators in equations (1) and (2). Because *SCtransform* and *SAttransform*, which are discussed further in Section 4, are symmetric, we do not need separate functions for their transposes. The *Htransform* and *calcHtransform* operators are sparse matrix-vector multiplies. All four functions return a vector.

Initial refactoring

The original version of the SAMURAI C++ code was a binary executable in a Docker container, and while Docker certainly facilitates execution on a laptop, its use complicated our code development work. For example, Docker makes the use of standard performance analysis tools difficult (or not possible), and the only container runtime available on the NCAR HPC system at the time due to security concerns was Charliecloud. Therefore, to facilitate performance evaluation, analysis, and improvement, as well as to enable access to more compute power and memory resources, we refactored SAMURAI to enable it to run on the NCAR HPC cluster. We first ported the Docker container version of the code to Charliecloud ([Priedhorsky and Randles 2017](#)), and then we ported the Charliecloud version to a standard HPC cluster: NCAR’s Cheyenne machine ([Computational and Information Systems Laboratory 2017](#)). This porting process included replacing the dependency on the Qt library ([The Qt Company 2021](#)), which was primarily used for strings, I/O, and time/date processing, with standard C++11

STL classes. Following that the CMake build structure was redesigned to simplify complex dependencies and to support multiple compilers (e.g., GNU, Intel, and PGI), as well as facilitate the use of standard performance analysis tools (e.g., GPROF, GPTL, and the Intel®VTune™ Profiler). This initial refactoring of SAMURAI made it easier to analyze its performance and determine where standard optimization approaches would be beneficial. In particular, we improved the performance of the code by restructuring the indexing, eliminating indirect addressing, and increasing thread parallelism, all of which are detailed in Section 4.

Experimental setup

In this brief section, we describe both the experimental platforms and the test problems that we use for evaluating performance. For the CPU results we use either a single node of the Cheyenne or Casper cluster at NCAR. A Cheyenne node contains two 18-core 2.3-GHz Intel Xeon E5-2697V4 Broadwell processors and either 64 or 128 GB/node of memory. A Casper node contains two 18-core 2.3-GHz Intel Xeon Gold 6140 Skylake processors with 384 GB/node of memory. Use of the large memory Casper nodes is required for one test problem that requires more memory than is available on Cheyenne. We compile all CPU code with the Intel 19.0.5 compiler. For the GPU results, we use a single NVIDIA V100 with 32 GB on the Casper data analysis and visualization cluster at NCAR, and compilation is done with the PGI 20.4 compiler.

For the purpose of examining the impact on the execution time of the code transformation, three example test case configurations were provided to us: SUPERCELL, HURRICANE, and 4-PANEL HURRICANE. These three idealized simulations were conducted by using the Weather Research and Forecasting (WRF) model (Skamarock et al., 2019) and resampled by a straight-and-level idealized flight with PAR (Precision Approach Radar) technology. Precision Approach Radar is the two-dimensional agile electronic scanning of the Active Electronically Scanned Arrays (AESA). Rapid scanning of the cloud volumes from PAR can capture the storm's evolution by returning the electronically steered beam more frequently.

The SUPERCELL test case represents simulated data from a type of thunderstorm called a "supercell." A supercell storm is likely to be associated with damaging winds and large hail, and sometimes severe tornadoes. The SUPERCELL test problem has a computational grid of size $241 \times 241 \times 33$ with 4.4 million observations. The aptly named second test case, HURRICANE, comes from a hurricane simulation. Hurricanes are well-known for bringing heavy rain, devastating winds, and storm surges that inflict severe damage to coastal areas or even inland areas. The HURRICANE test problem has a computational grid of size $105 \times 201 \times 33$ with 8.7 million observations. The test problem 4-PANEL

Table 2. Computational grid size and observation counts for the three test configurations.

Property		supercell	hurricane	4-panel hurricane
grid dimensions	idim	241	105	601
	jdim	241	201	401
	kdim	33	33	33
state size	nState	13.4 M	4.9 M	55.7 M
observations	mObs	4.4 M	8.7 M	29.7 M
non-zeros	nnz	584.4 M	1568.5 M	4608.1 M

HURRICANE was designed to have a similar data volume to the operational volume anticipated for APAR. Four panels were designed to be mounted on the left, right, bottom, and top of the aircraft, so that rapidly evolving features all around can be sampled by aircraft penetration into a hurricane. As compared to the HURRICANE case, the 4-PANEL HURRICANE configuration has a significantly larger state size and number of observations. A summary of the details of both the SUPERCELL, HURRICANE, and 4-PANEL HURRICANE test configurations is provided in Table 2. Note that 4-PANEL HURRICANE configuration presents several challenges that are not present with the SUPERCELL and HURRICANE configurations. Specifically, it requires a longer execution time and more memory than is typically available on the Cheyenne supercomputer. Therefore, this paper primarily provides analyses for the SUPERCELL and HURRICANE configurations. However, we do provide timing results for the 4-PANEL HURRICANE configuration at the end of Section 6.

In terms of computational cost considerations, recall that the number of observations determines the size of \mathbf{H} and \mathbf{H}^T and the computational grid size determines the dimensions of \mathbf{C} . Therefore, because the SUPERCELL case has somewhat larger computational grid dimensions (as indicated by the state size in Table 2) than the HURRICANE case, the cost the $SCtransform$ and $SAttransform$ operators will be a larger fraction of the total time. On the other hand, the HURRICANE problem has nearly double the number of observations of the SUPERCELL problem, and the cost of the $Htransform$ and $calcHTranspose$ operators are more dominant.

Code optimization

In Section 2, we introduced the four main operators used by SAMURAI. In this section, we describe some of the specific code optimization techniques that were performed on these operators. We first address the optimization of the two operators that perform pencil-type calculations, and then we discuss the matrix-vector product operators. Timing results are provided to illustrate the impact of these optimizations on the execution time.

Optimizing the pencil-calculation operators

Two of the SAMURAI operators (introduced in *SAMURAI* Section) can be categorized as pencil calculations: *SCtransform* and *SAtransform*. In other words, these operators first gather data from a full three-dimensional (3D) variable into a one-dimensional (1D) contiguous array (or pencil). Calculations are then performed on the 1D array, followed by a scattering of the modified data back into the full 3D variable. The specific calculation that occurs on the pencil is different for each of the operators. We next describe the specific changes that were made to the *SCtransform* operator, noting that the same techniques have been applied to both pencil operators.

Figure 2 contains a simplified representation of the *SCtransform* operator. *SCtransform* consists of an outer loop over the variables and three main inner loops that iterate over each pencil from the 3D grid-space. For simplification, we only include details for the k-pencil, as the code for the i- and j-pencils is similar. The basic structure of these pencil calculations is apparent from Figure 2. A gather from the full 3D-array *Astate* into a contiguous-pencil array *kTemp* occurs on lines 8–13. The method *filterArray()* performs a calculation on the *kTemp* array on line 15. Finally, a scatter from the *kTemp* array back into the full 3D-array *Cstate* occurs on lines 17–22. The values for *idim*,

jdim, and *kdim* are provided in Table 2, and the number of variables (*vDim*) is seven. Loop-level parallelism is achieved through the addition of an OpenMP directive on the outer loop (line 1). The fundamental issue with the code snippet in Figure 2 is that the operator is limited to 7-way parallelism due to the placement of the OpenMP directive. Note that in later sections, we refer to this original version of SAMURAI as the *BASE* version.

We address the lack of parallelism in the original version of the *SCtransform* operator by moving the OpenMP directives in one loop level and privatizing the contiguous pencil-array *kTemp* on which we perform the pencil calculation (i.e., *filterArray()* in this example). To enable execution on the GPU, we add OpenACC directives on the same loop as the OpenMP directive. This particular placement of the OpenMP and OpenACC directives prevents both OpenMP- and OpenACC-based parallelism from being used simultaneously. For OpenACC, we must also privatize the *idx* and *kTemp* variables and use the *collapse(2)* clause to allow for parallelization over both the *i* and *j* loops. The resulting *SCtransform* code is shown in Figure 3. Note that the detailed index calculation for *idx* has been replaced by a C++ macro and is indicated by *INDEX()*. A similar transform is performed for the remaining i- and j-pencil blocks of code.

```

1  #pragma omp parallel for private(idx)
2  for(int var=0; var<vDim; var++){
3    // Filter for K-pencil
4    real* kTemp = new real[kDim];
5    for(int iIdx=0; iIdx<iDim; iIdx++){
6      for(int jIdx=0; jIdx<jDim; jIdx++){
7        // Gather into contiguous array
8        for(int kIdx=0; kIdx<kDim; kIdx++){
9          idx=vDim*iDim*jDim*kIdx +
10             vDim*iDim*jIdx +
11             vDim*iIdx + var;
12          kTemp[kIdx] = Astate[idx];
13        }
14        // Perform calculation on pencil
15        kFilter->filterArray(kTemp, kDim);
16        // Scatter from contiguous array
17        for(int kIdx=0; kIdx<kDim; kIdx++){
18          idx=vDim*iDim*jDim*kIdx +
19             vDim*iDim*jIdx +
20             vDim*iIdx + var;
21          Cstate[idx] = kTemp[kIdx];
22        }
23      }
24    }
25    // Filter for J-pencil
26    ...
27    // Filter for I-pencil
28    ...
29  }

```

Figure 2. The original version of the *SCtransform* operator.

```

1  real* kTemp = new real[kDim];
2  ...
3  for(int var=0; var<vDim; var++){
4    // Filter for K-pencil
5    // Cstate = filter(Astate)
6    #pragma omp parallel for private(idx, kTemp)
7    #pragma acc parallel loop collapse(2)
8    #pragma acc private(idx, kTemp)
9    for(int iIdx=0; iIdx<iDim; iIdx++){
10     for(int jIdx=0; jIdx<jDim; jIdx++){
11       // Gather into contiguous array
12       for(int kIdx=0; kIdx<kDim; kIdx++){
13         idx=INDEX();
14         kTemp[kIdx] = Astate[idx];
15       }
16       // Perform calculation on pencil
17       kFilter->filterArray(kTemp, kDim);
18       // Scatter from contiguous array
19       for(int kIdx=0; kIdx<kDim; kIdx++){
20         idx=INDEX();
21         Cstate[idx] = kTemp[kIdx];
22       }
23     }
24   }
25   // Filter for J-pencil
26   // Cstate = filter(Cstate)
27   ...
28   // Filter for I-pencil
29   // Cstate = filter(Cstate)
30   ...
31 }

```

Figure 3. The optimized version of the *SCtransform* operator.

We refer to the version of the code with these optimized pencil calculations (in *SCtransform* and *SAttransform*) as *PENCIL-OPT*.

Optimizing the matrix-multiply operators

SAMURAI operators *Htransform* and *calcHTranpose* perform sparse matrix-vector multiplies with matrices H and H^T , respectively. The *Htransform* operator implements a standard multiplication by a compressed sparse row (CSR) matrix and is easily optimized by adding a single directive. However, optimizing the H^T matrix-vector product involves more significant code modifications, which we now describe.

The original version of *calcHTranpose* is shown in Figure 4 and consists of two loops: the outer over the total number of observations (*mObs*) and the inner over the number of columns in each row of observations. Because matrix H is stored in CSR format, the update to the vector *Astate* requires an indirect address. The indirect address for the store into *Astate* prevents both OpenMP threading for the CPU and OpenACC parallelization for the GPU. It is important to note that *obsVec* contains a large amount of physical information about an observation. For example, in addition to a physical value like temperature, it also contains when the observation was recorded along with other providence information. A consequence of this particular choice of data structure is that it results in a non-unit stride for the observation vector *obsVec*. Thus, for each cache-line that is loaded from memory, only a single word (i.e., *obsVec* [*mi*]) is utilized.

A revised version of the *calcHTranpose* operator is listed in Figure 5. This version of the code is quite a bit different from the original. The outer loop now iterates over vector *Astate*, and the inner loop is a reduction over all contributions to a particular state grid point. We create a new observation vector (*obsData* in line 13) which has unit stride and reduces the total size of the observational data that is accessed within *calcHTranpose*.

To access elements of matrix H (which is in CSR format), we introduce new arrays for indirect addressing: *mPtr*, *mVal*, and *I2H*. Finally, because the new version of *calcHTranpose* utilizes an indirect addressing approach for

```

1  for (int m=0; m<mObs; ++m) {
2    int mi = m*(7+varDim*derivDim)+1;
3    const int begin = IH[m];
4    const int end = IH[m + 1];
5    for(int j=begin; j<end; ++j) {
6      Astate[JH[j]] += H[j]*yhat[m]*obsVec[mi];
7    }
8  }

```

Figure 4. The original version of the *calcHTranpose* operator.

the load operations instead of for the store to *Astate*, it is now possible to parallelize the outer-loop with OpenMP and OpenACC directives. We refer to the version of the code that has both the previously described pencil optimizations as well as this new *calcHTranpose* operator as *H^T-OPT*.

Performance impact

Here we demonstrate the impact of the previously described code optimizations to the pencil and matrix-multiply calculations on our Cheyenne CPU platform (GPU results are discussed in Section 6). Figure 6 shows the execution time in seconds for the first twenty iterations of the SUPERCCELL case using three different versions of the SAMURAI code. Recall that the *BASE* refers to the original SAMURAI code, *PENCIL-OPT* contains the pencil operator optimizations,

```

1  #pragma omp parallel for
2  #pragma acc parallel loop gang vector &
3  #pragma acc reduction(+:tmp)
4  for(int n=0;n<nState;n++){
5    int ms = mPtr[n];
6    int me = mPtr[n+1];
7    real tmp = 0;
8    if(me>ms){
9      for(int k=ms;k<me;k++){
10         int m=mVal[k];
11         int j=I2H[k];
12         real val = yhat[m] * obsData[m];
13         tmp += H[j] * val;
14       }
15     }
16     Astate[n]=tmp;
17   }

```

Figure 5. The optimized version of the *calcHTranpose* operator.

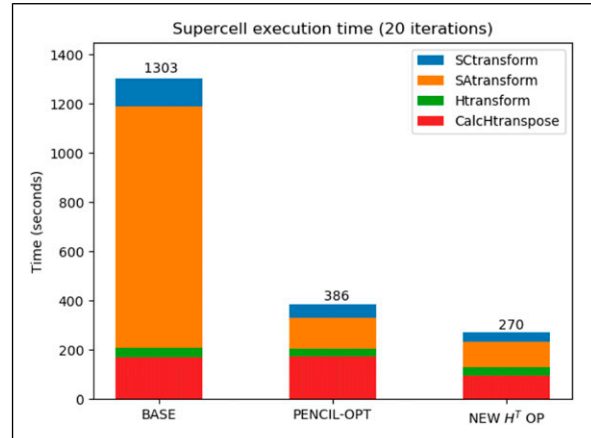


Figure 6. Execution time for 20 iterations of the SUPERCCELL configuration on a single node of Cheyenne using three different versions of SAMURAI. The total execution time in seconds is provided on top of each stacked bar.

and H^T -OPT contains both the pencil optimizations and the improved *calcHtranspose* operator. Figure 6 clearly demonstrates that the pencil-calculation optimization had a significant impact, reducing the execution time for twenty iterations of the SUPERCELL case from 1303 s to 386 s (more than a 3x speedup).

A more modest, although still significant, reduction was achieved by the *calcHtranspose* optimization, which further reduced the total time to 294 s. The combined impact of both optimizations reduces the execution time for the SUPERCELL configuration by a factor of 4.4.

The execution time in seconds for the first twenty iterations of the HURRICANE case using the same three versions of the SAMURAI code is provided in Figure 7. Note that for the HURRICANE configuration, the cost of the *Htransform* and *calcHtranspose* is significantly larger which reduces the overall impact of the *PENCIL-OPT* and H^T -OPT optimizations. The execution time is reduced from 558 to 326 s (a reduction of 1.7x).

Numerical optimization

Minimizing a cost function as in equation (1) is a common numerical optimization task (e.g., Nocedal and Wright (2006)), and a number of approaches are available depending on what operators are available and in what form. (Note that this optimization problem is unconstrained.) In this section, we first describe the current SAMURAI approach to minimizing (1). We then propose a more efficient approach and discuss convergence criteria and other efficiency factors. In the last subsection, we discuss a few other considerations for efficiency.

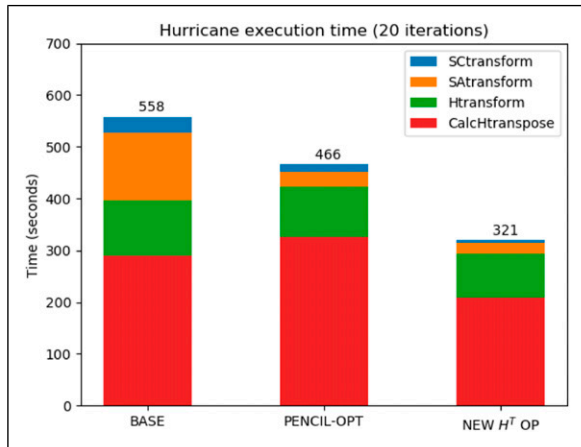


Figure 7. Execution time for 20 iterations of the HURRICANE configuration on a single node of Cheyenne using three different versions of SAMURAI. The total execution time in seconds is provided on top of each stacked bar.

Previous approach: Nonlinear conjugate gradient

As mentioned previously, many of the SAMURAI operators needed in the cost function are matrix-free. Therefore, SAMURAI uses a nonlinear Conjugate Gradient (NCG) method, as suggested in Gao et al. (2004), to minimize the cost function in (1) by determining the state \hat{x} that satisfies $\nabla J(\hat{x}) = 0$, where $\nabla J(\hat{x})$ is given in (2). In particular, NCG solves the following linear system

$$(\mathbf{I} - \mathbf{C}^T \mathbf{H}^T \mathbf{R}^{-1} \mathbf{H} \mathbf{C}) \hat{x} = \mathbf{C}^T \mathbf{H}^T \mathbf{R}^{-1} \mathbf{d}. \quad (4)$$

Nonlinear Conjugate Gradient methods are popular choices for large-scale nonlinear optimization problems for which matrices cannot be explicitly stored, as they have low memory costs and nice convergence properties. In SAMURAI, the NCG algorithm iteratively updates the estimate for \hat{x} , computing the optimal step length via a Brent's method line search and updating the search direction using the Polak-Ribière (PR) CG update parameter. (See Hager and Zhang (2006) for an overview of NCG methods.) From a software perspective, the NCG algorithm requires a function to evaluate both $J(\hat{x})$ and $\nabla J(\hat{x})$, for any given \hat{x} . As far as the computational cost of NCG in SAMURAI, the number of overall NCG iterations required for convergence is clearly key. At each iteration, the gradient must be evaluated, and a line search is performed as well. The line search is iterative and also requires function and gradient evaluations at each iteration. SAMURAI does not use a preconditioner for NCG.

Alternative approach: Truncated Newton

A natural question is whether we can reduce the solve time by either preconditioning NCG or using an alternative optimization algorithm. Although NCG is a logical choice when working with matrix-free operators, its use typically implies that we do not have information about the Hessian (the second derivative of the cost function). Lack of information about the Hessian means that determining an appropriate preconditioner for NCG is usually nontrivial (and remains an active area of research).

In the case of SAMURAI, the key to improving the optimization solver performance is the observation that an expression for the Hessian, $\nabla^2 J(\hat{x})$, easily follows from (2), as noted in Gao et al. (2004)

$$\nabla^2 J(\hat{x}) = (\mathbf{I} - \mathbf{C}^T \mathbf{H}^T \mathbf{R}^{-1} \mathbf{H} \mathbf{C}). \quad (5)$$

And while we cannot form the Hessian explicitly, we can easily obtain the matrix-vector (Matvec) products with the Hessian with a minor modification to the existing SAMURAI gradient routine. The addition of the Hessian Matvec operator to the SAMURAI code is significant as it broadens the choice of potential optimization algorithms to include a Newton approach, for example, which generally has better

convergence properties than NCG. As review, Newton's method is essentially a line search method of the form

$$\hat{x}_{k+1} = \hat{x}_k + \alpha_k \mathbf{d}_k, \quad (6)$$

where each iteration (k) steps closer (with step length α_k) to the solution in search direction \mathbf{d}_k . The search direction is chosen to be the so-called Newton direction that satisfies

$$\nabla^2 J(\hat{x}) \mathbf{d}_k = -\nabla J(\hat{x}) \quad (7)$$

(e.g., Nocedal and Wright (2006)). Solving for \mathbf{d}_k in the equation above requires inverting the Hessian, which is not practical for most applications (and in SAMURAI we do not have an explicit Hessian). However, because we now have the Hessian Matvec operator, the search direction can be determined from (7) by using an iterative linear solver. In optimization, this approach consisting of an outer Newton iteration and an inner iterative solver is referred to as a Truncated Newton (TN) solver (e.g., see Nash (2000)), as the inner iterative solver is stopped (or "truncated") at a given tolerance or iteration count.

The TN approach is particularly attractive for SAMURAI as the Hessian is symmetric positive definite, meaning that we can use Conjugate Gradient (CG) as the inner solver (referred to as TN-CG). Further, as pointed out in Gao et al. (2004), the Hessian (5) is well-conditioned as the identity matrix in the sum keeps the eigenvalues away from zero (a necessity for good CG convergence), negating the need for a preconditioner in the inner CG loop. As far as step length α_k in the outer iteration of TN-CG, in practice a line search calculates the optimal α_k , starting with an initial guess of $\alpha_k = 1$ (i.e., a full Newton step). We use the popular and robust Moré-Theunte line search method (Moré and Thuente 1994).

Further modifications

The NCG solver in SAMURAI is implemented directly in the C++ code (not an external package). However, to facilitate determining the best (i.e., fastest) unconstrained optimization solver for SAMURAI, we made extensive use of the PETSC software suite (Balay et al., 2019, 2020), including TAO (Toolkit for Advanced Optimization) in particular (Munson et al., 2012). The use of PETSC/TAO allows us to more quickly experiment with other NCG variants (e.g., by changing the search direction formula, line search method, or tolerances), as well as with TN (called Newton Line Search in PETSC/TAO) and its variants with little additional coding. Because the SAMURAI developers wanted dependence on PETSC to be optional, we eventually added the TN-CG algorithm directly to the SAMURAI code as well (including an implementation of Moré-Theunte). All timings presented later in this paper are for the SAMURAI

TN-CG implementation (not PETSC) unless otherwise noted.

Another code improvement in terms of solver performance was to combine the SAMURAI function and gradient evaluation codes. In other words, we added a third function to return the evaluations of both the gradient and function. This combination eliminates the repetitive computation necessary for both evaluations when done separately, particularly affecting the NCG solver (as will be seen in Table 3).

Impact of the solver choice on operator calls

We now discuss the impact of using the TN-CG solver (instead of NCG) on the execution time of SAMURAI. In Table 3, we list six functions that constitute the bulk of the time in the two nonlinear solvers. For each of these functions, we list the number of calls to each of the four fundamental operators that were optimized as discussed in Section 4. Note that the two-line search functions (bottom two rows) are iterative, and the operator counts are per iteration. For comparison, pseudocode for the NCG and TN-CG solvers is listed in Figures 8 and 9, respectively.

For NCG, Figure 8 indicates that the expense of each NCG iteration can be attributed to the *FuncGradient* and *BrentLineSearch* calls. For each of our two test cases, we find that the average number of iterations per line search call is approximately three. Therefore, Table 4 lists the total operator calls per NCG iteration in the first row assuming three iterations for the line search.

For the TN-CG solver, Figure 9 shows that the primary cost of the inner CG iteration is dominated by the call to *FuncHessian*, which for SAMURAI is roughly the same cost as a call to *FuncGradient* (as shown Table 3). Table 4 lists the total operator calls per inner CG iteration for TN-CG in the bottom row. The outer loop of the TN-CG solver consists of the cost of the inner iterative solver (which depends on the number of iterations) as well as the line search (*MTLineSearch*) listed in the last row of Table 3. Because the outer loop of the TN-CG solver (the Newton step) requires an iterative line search, the computation in the outer loop iteration (excluding the inner iterations) is typically more expensive than each inner CG iteration, which is important to keep in mind when minimizing the computational cost. Naturally, the rate of convergence of the outer Newton iteration is closely related to the accuracy to which the inner CG iteration is solved. In other words, if a "better" search direction is found via more inner iterations, then fewer Newton steps are likely needed. Balancing the trade-off between the inner and outer iterations is important for using a TN-CG solver efficiently and is controlled by the selection of the tolerances for both the outer Newton iteration (ϵ) and for the inner CG iteration (ϵ_{CG}). We further discuss convergence tolerances in the next subsection.

Table 3. The number of operator calls (for the four identified fundamental operators) needed for functions in NCG and TN. Note that the `BrentLineSearch()` is calling `funcValue()` and `funcGradient()` and the `MTLineSearch()` is calling `funcValueAndGradient()` at each iteration (indicated by a *).

Function name	Operation	<i>SCtransform</i>	<i>SAttransform</i>	<i>Htransform</i>	<i>calchHTranspose</i>
<code>funcValue(x_k)</code>	$J(\mathbf{x}_k)$	1	1	1	0
<code>funcGradient(x_k)</code>	$\nabla J(\mathbf{x}_k)$	2	2	1	1
<code>funcValueAndGradient(x_k)</code>	$J(\mathbf{x}_k)$ and $\nabla J(\hat{\mathbf{x}}_k)$	2	2	1	1
<code>funcHessian(x_k, y)</code>	$(\nabla^2 J(\mathbf{x}_k))\mathbf{y}$	2	2	1	1
<code>BrentLineSearch(x_k, d_k)</code>	$\min_{\alpha>0} J(\mathbf{x}_k + \alpha \mathbf{d}_k)$	3*	3*	2*	1*
<code>MTLineSearch(x_k, d_k)</code>	$\min_{\alpha>0} J(\mathbf{x}_k + \alpha \mathbf{d}_k)$	2*	2*	1*	1*

```

Initial guess: x0
Initial cost function: J(x0) = FuncValue(x0)
d0 = -g0 = -FuncGradient(x0)
for (k=0; k < max_ncg_its; k++) {
    αk = BrentLineSearch(xk, dk) \\iterative
    xk+1 = xk + αkdk
    gk+1 = FuncGradient(xk)
    Check Convergence \\ exit?
    βk calculation \\CG parameter (PR formula)
    dk+1 = -gk+1 + βkdk \\
}

```

Figure 8. Pseudocode for the original nonlinear Conjugate Gradient (NCG) algorithm in SAMURAI. The cost in terms of operator calls for the functions in green are listed in Table 3.

```

Initial guess: x0
Initial cost function: J(x0) = FuncValue(x0)
for (k=0; k < max_newton_its; k++) {
    gk = -∇J(xk) \\FuncGradient
    \\ find dk s.t. ∇2J(xk)dk = gk (standard CG)
    CG initial guess\residual: d0 = 0; r0 = gk
    CG initial basis vector: p0 = r0
    for (i=1; i < max_cg_its; i++) {
        calculate: βi, pi \\ CG vector ops
        FuncHessian(xk, pi)
        calculate: αi, di+1, ri+1 \\CG vector ops
        Check CG convergence \\ exit CG loop?
    }
    αk = MTLineSearch(xk, dk) \\iterative
    xk+1 = xk + αkdk \\take newton step
    Check Newton convergence \\ exit?
}

```

Figure 9. Pseudocode for the Truncated-Newton (TN) algorithm in SAMURAI with CG for the inner solver. The cost in terms of operator calls for the functions in green are listed in Table 3.

Convergence criteria

Convergence criteria are important as they directly affect the number of solver iterations and, therefore, the overall solve time. The original SAMURAI NCG solver checks for convergence with a step-size criterion at iteration k by

comparing the difference in cost function value in consecutive NCG steps to a fixed tolerance

$$2.0 \frac{\left| J(\hat{x}_k) - J(\hat{x}_{k-1}) \right|}{\left(\left| J(\hat{x}_k) \right| + \left| J(\hat{x}_{k-1}) \right| \right)} < \epsilon, \quad (8)$$

where $\epsilon = 1e^{-5}$. As part of reworking the solvers, we proposed a more common convergence criteria, which checks the relative reduction in the gradient of the cost function

$$\frac{\left\| \nabla J(\hat{x}_k) \right\|}{\left\| \nabla J(\hat{x}_0) \right\|} < \epsilon, \quad (9)$$

where ϵ is a user-specified parameter. This second criterion is certainly more widely used (e.g., in packages like PETSC-TAO) and arguably more intuitive, making it easier to compare across various solvers and test problems. In practice, we found that when the original convergence criterion (8) is met for $\epsilon = 1e^{-5}$, the relative reduction in the gradient in (9) may be an order or two in magnitude larger, depending on the problem. For this reason, we have set the current default in SAMURAI for the relative gradient criteria in (9) to $\epsilon = 1e^{-4}$.

As already noted, the TN-CG solver requires a tolerance for the inner CG iteration (ϵ_{CG}). For the SAMURAI application, matvecs with the Hessian are approximately the same cost as evaluating the gradient (e.g., Table 3), which means that the cost of an inner CG iterations is reasonable—and less than the cost of an NCG iteration (e.g., Table 4). Furthermore, a suitable convergence tolerance for ϵ in SAMURAI is not that small (as compared with other numerical applications which may require more accurate solutions). As a result, in our experience thus far, the fastest times to solution for TN-CG on a number of test problems have been when $\epsilon_{CG} = \epsilon$, which results in a single Newton step. As noted previously, choosing the default tolerance ϵ for checking the reduction of the gradient is of computational importance, and it is important to keep in mind that

convergence is seldom linear and tends to level off (i.e., each additional order of magnitude requested requires more iterations than the previous). As an example, Table 5 lists the number of inner CG iterations for the TN-CG solver (with $\epsilon_{CG} = \epsilon$) on our two primary test cases for three tolerance levels, and increases in run time (not listed) are proportional to the increases in iterations. For the HURRICANE test problem in particular, the cost of decreasing the convergence tolerance by an order of magnitude is nontrivial.

Although the information in Tables 3 and 4 provides some insight on the costs per iteration, we cannot estimate the impact of using TN-CG versus NCG without knowing the required iterations for convergence, which is of course problem dependent. Therefore, for our two primary test problems, Table 6 lists the speedup of TN-CG over NCG. These results were obtained with the PETSc implementations of the two solvers to ensure that the solver effects were isolated. In other words, by comparing PETSc implementations (as opposed to the native SAMURAI implementation), we were able to compare the solvers using the same exact convergence criteria, line search method, and (combined) function and gradient evaluations. Table 6 indicates a speedup of approximately 4x for both problems due to using the TN-CG solver.

Correctness

Verifying code correctness due to the optimizations (including the solver change) was done using an analytic 3D Beltrami flow problem with alternating counter-rotating vortices and updrafts using input pseudo-observations of u , v , and w following Shapiro et al. (2009). The Beltrami flow is an analytic wind field that would be obtained in a similar manner to the test cases used for performance

validation. The main difference between the analytic 3D Beltrami flow solution and the SAMURAI retrieval is that a mass continuity equation is included in SAMURAI for the wind retrieval. The mass continuity equation enforces the mass-weighted vertical flow to be physically consistent with the divergence of the mass-weighted horizontal flow. This constraint is required for most meteorological applications because vertical velocity is a rare direct observation, so an additional assumption is usually required to retrieve a full 3D wind field. Since the analytic equation does not have this constraint, a slight difference between an analytic solution and SAMURAI solution can be expected. Because bitwise reproducibility is not required for SAMURAI due to inherent uncertainties in the observations and analysis, computing the root-mean-square error (RMSE) between the analytic Beltrami solution and the SAMURAI analysis is a more meaningful assessment. The gradient terms (vorticity and divergence) are calculated because the accuracy of vorticity and divergence fields are indirect wind observations but are important wind variables to understand the atmospheric dynamics. Therefore, we calculated both the RMSE and the related root-mean-square percentage error (RMSPE) for five key variables for both the original and optimized version of SAMURAI for the Beltrami problem. (Note that RMSPE is simply the root-mean-square relative error as a percentage.) For reference, Table 7 lists the RMSE and RMSPE statistics for each field using the optimized SAMURAI code. These statistics are nearly identical to those from the original software version (before optimization) and are well below the error tolerance for the physical interpretation of the results. Less stringent tests with thermodynamic scalar variables produce similar statistics (not shown).

We note that numerical comparisons of the output from the SUPERCELL and HURRICANE test cases also show very

Table 4. The number of operator calls (for the four identified fundamental operators) needed for one iteration of NCG and one inner iteration of TN-CG, assuming an average of three calls to *BrentLineSearch* per NCG iteration.

Solver	<i>SCtransform</i>	<i>SAttransform</i>	<i>Htransform</i>	<i>calcHTranspose</i>
NCG iteration	11	11	7	3
TN-CG inner iteration	2	2	1	1

Table 5. Number of CG iterations in the inner TN solver for various convergence tolerances (ϵ) in (9).

Test case	ϵ	TN inner CG iterations
SUPERCELL	$1e^{-3}$	309
	$1e^{-4}$	592
	$1e^{-5}$	889
HURRICANE	$1e^{-3}$	107
	$1e^{-4}$	549
	$1e^{-5}$	1243

Table 6. For our two primary test cases, we list the speedup of TN-CG over NCG in the rightmost column. These results were obtained with the PETSc implementations of these solvers in SAMURAI with a convergence tolerance of $\epsilon = 1e^{-5}$ (and $\epsilon_{CG} = \epsilon$ for TN-CG). Iteration counts are also listed for reference.

Test case	PETSc NCG iterations	PETSc TN-CG inner iterations	TN-CG speedup
SUPERCELL	889	913	4.0x
HURRICANE	1243	1321	3.7x

Table 7. Root mean square error (RMSE) and root mean square percentage error (RMSPE) comparing an analytic three-dimensional Beltrami configuration to the optimized SAMURAI code analysis.

Variable	units	RMSE	RMSPE, %
zonal wind	m/s	0.034	0.0065
meridional wind	m/s	0.034	0.0065
vertical wind	m/s	0.039	0.0061
vorticity	1/s	$7.148e^{-5}$	0.019
divergence	1/s	$8.755e^{-5}$	0.068

small differences (not shown) that are well below the uncertainty of the expected analysis using real observations. These RMSE statistics calculated from the original and optimized versions are very low and comparable, which provides further reassurance that optimized version of SAMURAI produces correct results.

Experimental Results

In this section, we first demonstrate the combined impact that the code optimization (Section 4) and new solver (Section 5) have on the overall SAMURAI execution time on both CPU and GPU platforms for our two primary test configurations. Then we present CPU results for the larger 4-PANEL HURRICANE configuration that more closely matches the anticipated size of the APAR data and discuss its associated challenges.

SUPERCCELL and HURRICANE test configurations

Recall from Figure 6 that the initial code optimizations reduced the time to perform twenty iterations of the SUPERCCELL configuration from 1303 to 270 s. The left bar in Figure 10 indicates that use of the TN-CG (instead of NCG) further reduces execution time to 40.6 s on the CPU, which represents a 6.6x speedup on the CPU.

In addition, Figure 10 also includes the execution time for TN-CG method on a single GPU. On the GPU the execution time is further reduced to 15.4 s, which is an overall speedup of nearly 2.6x versus the CPU version. The speedup is different for each particular operator and ranges from 1.6x to 3.9x. The greatest speedup is observed for the *calcHtranpose* operator which accesses significantly larger data structures than the pencil operators. The achieved speedup directly results from the significant improvement in the main memory bandwidth that is possible on the GPU. In particular, the STREAM benchmark for Intel Broadwell nodes is 153.6 GB/sec while for the NVIDIA V100, it is 900 GB/sec.

Figure 11 provides a timing information for 20 iterations of the HURRICANE configuration. For the HURRICANE configuration, the relative importance of the cost of *Htransform* and

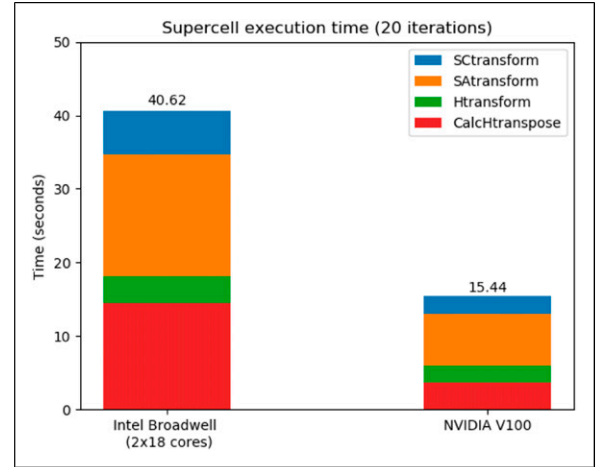


Figure 10. Execution time for twenty iterations of SUPERCCELL configuration using TN-CG solver on CPU (left bar) and GPU (right bar) platforms.

calcHtranpose is much larger as they contribute to 90.9% of the total time for versus 44.7% of the time SUPERCCELL configuration. The execution time for twenty iterations is 52.4 s on the CPU and 17.4 s on the GPU, and the speedup that the GPU enables is a larger 3.0x versus the CPU version for the HURRICANE test. As with the SUPERCCELL configuration, the speedup enabled by use of the GPU is most pronounced for *calcHtranpose* operator, which accesses very large data structures.

We now summarize the collective impact of the code optimizations, numerical solver changes and the GPU execution on the execution time of SAMURAI for both the SUPERCCELL and HURRICANE test cases in Table 8. The total run time for the solver component of SAMURAI to convergence is provided and does not include any of the initialization or I/O costs. For the SUPERCCELL configuration, SAMURAI originally required 577 min to converge. The time to converge has been reduced to 17 min on a single CPU node, and 5.4 min on a single GPU. Our efforts resulted in a 34x speedup on CPU and a 107x speedup using a GPU. For the HURRICANE configuration the original execution time of 609 min was reduced to 19 min on a single CPU node and 4.7 min on a single GPU. This represents a similar 32x speedup on CPU and a 130x speedup on a single GPU.

Note that the results in Table 8 are timings to convergence. Recall from Section 5 that for the original SAMURAI NCG solver, convergence is met when the criterion in (8) is satisfied for $\epsilon = 1e^{-5}$. The new TN-CG solver, however, declares that convergence is met when the criterion in (9) is satisfied for $\epsilon = 1e^{-4}$. Although the two different criteria do not result in the same quality of solution (as measured by the relative reduction of the gradient, e.g.), both produce a solution of acceptable quality, and comparing in this manner replicates how application users will run the code. Alternatively, one could choose to evaluate the speedup without factoring in the influence of the different convergence

criteria by modifying the tolerances for TN-CG to be equivalent to the measured relative gradient reduction when the original criterion in (8) is satisfied for $\epsilon = 1e^{-5}$. In other words, when SUPERCELL converges with NCG with the original criterion, we have that $\|\nabla J(\hat{x}_k)\| / \|\nabla J(\hat{x}_0)\| = 4.3e^{-4}$. We therefore can run TN-CG using the criterion in (9) with $\epsilon = 4.3e^{-4}$, which will result in fewer TN-CG iterations and even greater speedup. Similarly, for the HURRICANE problem, we would use the criterion in (9) with $\epsilon = 4.5e^{-5}$. In this case, though, TN-CG will require more iterations to convergence, resulting in a slightly lower overall speedup.

Considering a larger problem

We now discuss the 4-PANEL HURRICANE configuration that was designed to have a similar data volume to the

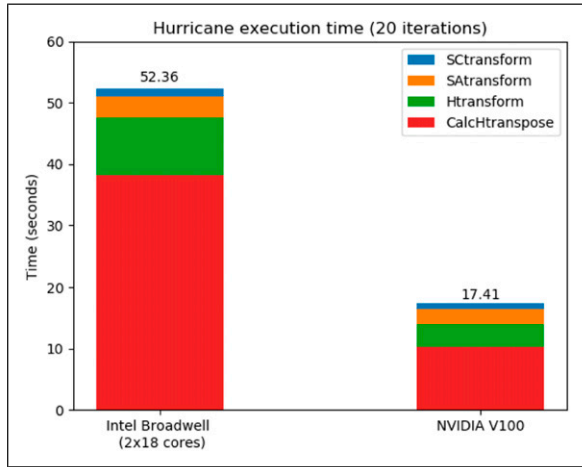


Figure 11. Execution time for 20 iterations of HURRICANE configuration using TN-CG solver on CPU (left bar) and GPU (right bar) platforms.

Table 8. Summary of SAMURAI code optimizations and corresponding timings to convergence. Code versions are defined in Section 4, and CPU and GPU platforms details are given at the beginning of Section 3.

Code version	Solver	Platform	Execution time (min)	
			SUPERCELL	HURRICANE
BASE	NCG	CPU	577	609
PENCIL-OPT	NCG		151	382
NEW H^T OPT	NCG		109	258
NEW H^T OPT	TN-CG		17	19
NEW H^T OPT	NCG	GPU	53	72
	TN-CG		5.4	4.7
Speedup (original CPU/final CPU)			34x	32x
Speedup (original CPU/final GPU)			107x	130x

operational volume anticipated for APAR. As compared to the HURRICANE case, this 4-PANEL HURRICANE configuration has a state size that is $\sim 11x$ larger and has $\sim 3.4x$ more observations. Recall that the exact problem specifications are listed in Table 2.

Several challenges exist in terms of running the 4-PANEL HURRICANE configuration. First, a total of 213 GB of memory is required, which exceeds the 128 GB of memory that is available on Cheyenne. Therefore, we use NCAR's Casper machine, which is a heterogeneous system of specialized data analysis and visualization resources that contains multiple large-memory nodes. Second, the problem currently requires more memory than is typically available on a single GPU, and we do not currently have the ability to utilize multiple GPUs. Although we did attempt to execute the 4-PANEL HURRICANE configuration on a single GPU using managed memory, the execution time was catastrophically slow. The code failed to even completing a single iteration of the solver after 6 hours. Note that the average execution time for a single iteration of the TN-CG solver for the 4-PANEL HURRICANE configuration on the Casper CPU node is 11 seconds. The detailed execution times for the 4-PANEL HURRICANE configuration on the large memory Casper nodes are provided in Table 9. Recall that the initial goal of this project was to allow the processing of the APAR observational data in less than 12 hours, which is the queue limit of the NCAR supercomputer Cheyenne. The execution times (in both minutes and hours) for the 4-PANEL HURRICANE configuration on the large memory Casper nodes are provided in Table 9. For the BASE version of the code, the starting point for this effort, an execution time of 43.4 hours clearly exceeds the desired 12 hours. Furthermore, despite the optimizations introduced in the PENCIL-OPT and NEW H^T OPT versions of the code, the NCG solver still takes 19.6 and 12.6 hours of execution time, respectively. However, with the addition of the TN-CG solver, the execution time for the 4-PANEL HURRICANE configuration is reduced to 106 minutes or 1.8 hours, which is a reduction in

Table 9. Execution times for the larger 4-PANEL HURRICANE configuration on a large memory CPU node on NCAR's Casper machine. (Memory constraints prevented execution on the GPU.)

Code version	Solver	Platform	Execution time	
			(min)	(hours)
BASE	NCG	CPU	2606	43.4
PENCIL-OPT	NCG		1175	19.6
NEW H^T OPT	NCG		759	12.6
NEW H^T OPT	TN-CG		106	1.8
Speedup (original CPU/final CPU)			25x	

execution time of 25x versus the original BASE code on the CPU.

Concluding remarks

We have described our efforts to modernize and create a portable performant version of SAMURAI. This effort involved the porting SAMURAI to the NCAR high-performance computing environment, multiple rounds of code optimization, integration of an improved numerical solver, and the porting to GPU through the use of OpenACC directives. These efforts had a profound impact on overall execution time achieving a 30x speedup on a single CPU based node and a greater than 100x speedup on a single v100 GPU node. Our transformations will enable the processing of the anticipated full resolution APAR data sets using commonly available HPC resources. It will also now be possible to use basic desktop resources to process modest resolution data. Finally, the GPU enablement will allow SAMURAI to be utilized in flight, potentially allowing for the ability to adjust science goals in real-time.

Despite the progress made towards reducing the execution time of the solver within SAMURAI, additional work is still necessary. Specifically, while the SUPERCCELL and HURRICANE achieve excellent speedup on the GPU, it is not currently possible to execute the 4-PANEL HURRICANE configuration on the GPU due to memory limitations of the GPU. We therefore intend to create a distributed memory version of SAMURAI to enable the use of both lower-memory CPU nodes and multiple GPUs. Additionally, with the significant acceleration of the solver component of SAMURAI, the cost of I/O in both the initialization and finalization stages are now upwards of 30% of the total run time. Consequently, improvements to the I/O subsystem will need to be considered for future optimization efforts.

Acknowledgements

We would like to acknowledge high-performance computing support from Cheyenne (doi:10.5065/D6RX99HX) provided by NCAR's Computational and Information Systems Laboratory, sponsored by the National Science Foundation. MMB and TYC acknowledge support from NSF award OAC-1661663. We would also like to thank Scott Ellis and Wen-chau Lee of NCAR for supporting the SAMURAI optimization effort and Mike Dixon of NCAR for his software and build support.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the Office of the Director (OAC-1661663).

ORCID iDs

John M Dennis  <https://orcid.org/0000-0002-2119-8242>
 Allison H Baker  <https://orcid.org/0000-0003-2436-7838>
 Brian Dobbins  <https://orcid.org/0000-0002-3156-0460>
 Jian Sun  <https://orcid.org/0000-0002-6987-8052>
 Ting-Yu Cha  <https://orcid.org/0000-0002-6292-8483>

References

- Balay S, Abhyankar S, Adams MF, et al. (2019) PETSc Web page. <https://www.mcs.anl.gov/petsc>
- Balay S, Abhyankar S, Adams MF, et al. (2020) *PETSc Users Manual*. Technical Report ANL-95/11 - Revision 3.14, Argonne National Laboratory. <https://www.mcs.anl.gov/petsc>
- Bell MM, Montgomery MT and Emanuel KA (2012) Air and sea enthalpy and momentum exchange at major hurricane wind speeds observed during CBLAST. *Journal of the Atmospheric Sciences* 69(11): 3197–3222.
- Boehm AM and Bell MM (2021) Retrieved thermodynamic structure of Hurricane Rita (2005) from airborne multi-Doppler radar data. *Journal of the Atmospheric Sciences* 78(5): 1583–1605. DOI: [10.1175/JAS-D-20-0195.1](https://doi.org/10.1175/JAS-D-20-0195.1)
- Computational and Information Systems Laboratory (2017) Cheyenne: SGI ICE XA Cluster. DOI: [10.5065/d6rx99hx](https://doi.org/10.5065/d6rx99hx). DOI: [10.5065/D6RX99HX](https://doi.org/10.5065/D6RX99HX).
- Foerster AM and Bell MM (2017) Thermodynamic retrieval in rapidly rotating vortices from multiple-Doppler radar data. *Journal of Atmospheric and Oceanic Technology* 34(11): 2353–2374. DOI: [10.1175/JTECH-D-17-0073.1](https://doi.org/10.1175/JTECH-D-17-0073.1)
- Foerster AM, Bell MM, Harr PA, et al. (2014) Observations of the eyewall structure of Typhoon Sinlaku (2008) during the transformation stage of extratropical transition. *Mon. Wea. Rev* 142: 3372–3392.
- Gao J, Xue M, Brewster K, et al. (2004) A three-dimensional variational data analysis method with recursive filter for Doppler radars. *Journal of Atmospheric and Oceanic Technology* 21(3): 457–469.
- Hager WH and Zhang H (2006) A survey of nonlinear conjugate gradient methods. *Pacific Journal of Optimization* 2: 35–58.
- Jorgensen D, Matejka T and DuGranrut J (1996) Multi-beam techniques for deriving wind fields from airborne Doppler radars. *Meteor. Atmos. Phys* 59: 83–104. DOI: [10.1007/BF01032002](https://doi.org/10.1007/BF01032002)
- Moré JJ and Thiente DJ (1994) Line search algorithms with guaranteed sufficient decrease. *ACM Transactions on Mathematical Software* 20(3): 286–307. DOI: [10.1145/192115.192132](https://doi.org/10.1145/192115.192132)
- Munson T, Sarich J, Wild S, et al. (2012) TAO 2.0 users manual. Technical Report ANL/MCS-TM-322, Mathematics and Computer Science Division, Argonne National Laboratory. <http://www.mcs.anl.gov/tao>
- Nash SG (2000) A survey of truncated-Newton methods. *Journal of Computational and Applied Mathematics* 124(1): 45–59.
- Nocedal J and Wright SJ (2006) *Numerical Optimization*. second edition. New York, NY, USA: Springer.

- Ooyama KV (2002) The cubic-spline transform method: basic definitions and tests in a 1D single domain. *Monthly Weather Review* 130(10): 2392–2415. DOI: [10.1175/1520-0493\(2002\)130<2392:TCSTMB>2.0.CO;2](https://doi.org/10.1175/1520-0493(2002)130<2392:TCSTMB>2.0.CO;2)
- Priedhorsky R and Randles T (2017) *Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC. SC '17*. New York, NY, USA: Association for Computing Machinery. DOI: [10.1145/3126908.3126925](https://doi.org/10.1145/3126908.3126925)
- Shapiro A, Potvin CK and Gao J (2009) Use of a vertical vorticity equation in variational dual-Doppler wind analysis. *Journal of Atmospheric and Oceanic Technology* 26(10): 2089–2106. DOI: [10.1175/2009JTECHA1256.1](https://doi.org/10.1175/2009JTECHA1256.1). https://journals.ametsoc.org/view/journals/atot/26/10/2009jtecha1256_1.xml
- Skamarock WC, Klemp JB, Dudhia J, et al. (2019) *A Description of the Advanced Research WRF Version 4*. Technical Report NCAR/TN-556+STR, National Center for Atmospheric Research. DOI: [10.5065/1dfh-6p97](https://doi.org/10.5065/1dfh-6p97)
- The Qt Company (2021) Qt framework. <https://www.qt.io/>. (Accessed 19 April 2021).
- Vivekanandan J and Loew E (2018) Airborne polarimetric Doppler weather radar: trade-offs between various engineering specifications. *Geoscientific Instrumentation, Methods and Data Systems* 7(1): 21–37. DOI: [10.5194/gi-7-21-2018](https://doi.org/10.5194/gi-7-21-2018). <https://gi.copernicus.org/articles/7/21/2018/>

Author biographies

John M. Dennis received a PhD in computer science in 2005. He is a Scientist III in the Computer Information and Systems Laboratory at the National Center for Atmospheric Research. He co-leads a research group that focuses on improving the ability of large-scale geoscience applications to utilize current and future computing platforms. His research interests include parallel algorithm and compiler optimization, graph partitioning, and data-intensive computing.

Allison H. Baker received her PhD in Applied Mathematics from the University of Colorado in 2003. She is a Project

Scientist III in the Computational and Information Systems Laboratory at the National Center for Atmospheric Research (NCAR). Her research interests include high-performance computing, numerical linear algebra, performance analysis, data compression, and model verification.

Brian Dobbins has a master's degree in high-performance computing from the University of Edinburgh and is a software engineer in NCAR's Computational and Information Systems Lab, working on application performance and future architectures.

Michael M. Bell received a PhD in Meteorology from the Naval Postgraduate School in 2010. He is currently a Professor at Colorado State University. His primary interests are in tropical, mesoscale, and radar meteorology.

Jian Sun is a software engineer from the Application Scalability And Performance Group (ASAP) at National Center For Atmospheric Research (NCAR). His research interests include porting the components of an earth system model to GPU and optimizing their performance. Jian received his Ph.D. degree from University of Tennessee, Knoxville.

Youngsung Kim received a BS degree in electrical engineering from Dankook University, South Korea in 1995, and an MS degree in computational science and engineering from the University of Utah in 2012. He is a software performance engineer and tool developer interested in supporting domain scientists through technical consultancy. He is a member of the performance group for the Energy Exascale Earth System Model at Oak Ridge National Laboratory.

Ting-Yu Cha is a Ph.D. candidate at Colorado State University. Her interests include researching the mesoscale structure, evolution, and precipitation of tropical cyclones with ground-based and airborne radar data.