NASA GPU Hackathon Yields Significant Code Improvements

The NASA GPU Hackathon 2020 brought together application developers and computer experts to help get important NASA applications running effectively on graphics processing unit (GPU) nodes. Nine teams of application developers participated in this virtual event, a major impetus for teams to modernize codes of interest for NASA missions to CPU nodes containing GPU accelerators, with a focus on hands-on problem solving. The photo in Figure 1 shows 30 of the more than 50 participants. The HECC project and NVIDIA jointly organized the event, and HECC provided five Pleiades nodes each with 4 V100 GPUs for teams to use.

Figure 1: Virtual Hackathon group photo.

The virtual event, which took place over four days from September 28–October 7, 2020, used Microsoft Teams and Slack as collaboration tools. Each team consisted of three to six members from NASA Centers and supporting organizations. The teams were paired with one to two mentors from industry, government, and academia. The experience levels of the teams ranged from being GPU novices to advanced CUDA programming experts. OpenACC and the emerging Kokkos API were used in addition to CUDA for GPU programming.

During the event, which focused on accelerating AeroSciences and CFD applications, most teams achieved considerable performance improvements on both GPUs and CPUs. For example, a team with no GPU experience completed a first port of a time-critical loop to a GPU. Another team of expert CUDA programmers were able to restructure their algorithm, yielding a factor of five speed-up. And another team sped up some of their CUDA kernels by a factor of 20, which directly translated into their production code.

This article highlights some of the many successes resulting from the event.

Finite-Element Methods Reaching for the Accelerator Roofline using CUDA

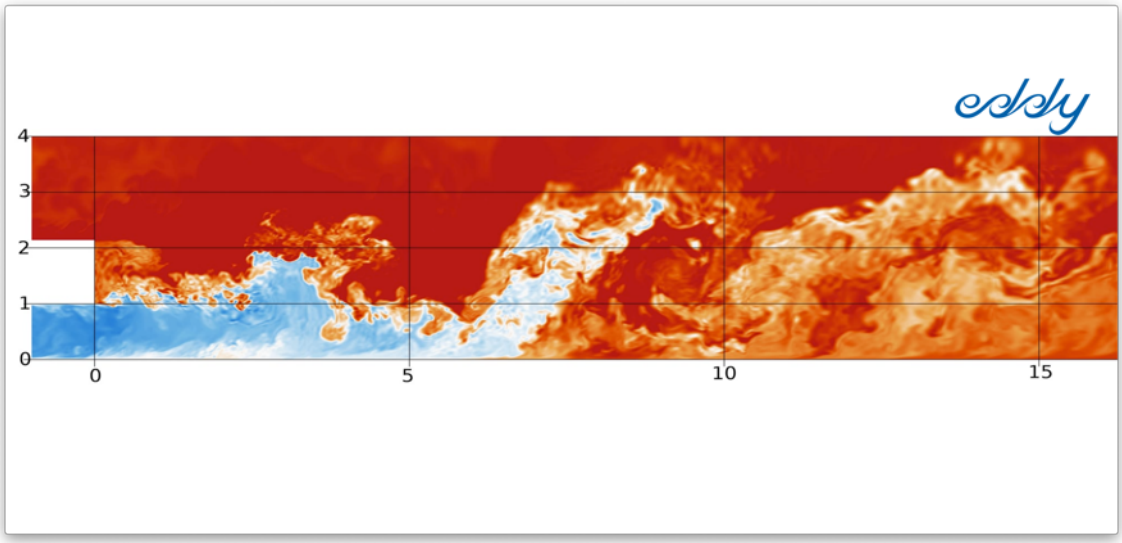


Figure 2: Temperature contours from a film-cooling benchmark computed using *eddy*.

Finite-element methods (FEM) are popular due in part to their high arithmetic intensity, low communication costs, and potential for arbitrary order of accuracy. The *eddy* mini-app implements the core of a space-time finite-element method in C/C++/CUDA. Both the full application and the mini-app run in arbitrary polynomial degree (N) in both space and time, but N=2, 4, 8, 16 are typically used due to the ease of mapping to CPU/GPU hardware. The image in Figure 2 results from a full application simulation using a combination of elements of polynomial degree from 2 through 16, depending on the local error. Team *eddy* started from a pre-hackathon CUDA C++ implementation of the mini-app, which was developed following the CPU code and CUDA best practices. The mini-app represents the majority of the execution time of the code, and is composed of three steps: interpolation to quadrature points in both space and time, computation of the flux at quadrature points, and accumulation of the residual within each element. The initial mini-app mimicked the CPU implementation, where these steps are performed in separate kernels that read/write from/to main memory.

The initial mini-app was already performing fairly well, especially at high order (N=8, 16), and was using a significant percentage of the roofline floating-point performance. However, the overall algorithm was bandwidth (BW) limited and the thread blocks were too small for lower-order polynomials. In CUDA a *thread block* is a programming abstraction that represents a group of threads that can be executed serially or in parallel.
Initially, the hope was to determine a new approach to the kernels to ease the BW limitations. After a thorough code review the team came up with the following steps to improve the lower-order cases:

- Exposing extra parallelism: the initial version optimized thread usage for the interpolation, however it was found that using a thread structure that was optimal for the remaining two kernels performed better, even with some idle threads during the interpolation step. The strategy provided a speedup of 33% over the initial implementation at N=2 and N=4;
- Taking advantage of the space-time nature of the algorithm, they fused the 3 compute kernels into one. This decoupled the work on the spatial degrees of

freedom performed in each temporal slice and also increased parallelism.

- Increasing the use of dynamic shared memory for some of the most frequently accessed arrays, which provided an additional 40% speedup for the lowest order case, N=2.

By combining the modifications, the team achieved a 2X speedup at 2nd order, and about 10% improvement at 4th order, even though the algorithms still remained memory bound. The group benefited greatly from this exercise, and are now testing similar ideas both on the higher-order implementations, and on the newest NVIDIA A100 card with increased shared memory capabilities per streaming multiprocessor.

Order	TFLOPS BEFORE	BlockSize
2	0.431	4
4	0.901	16
8	1.357	64
16	1.472	256
Order	TFLOPS AFTER	BlockSize
2	0.787	16
4	0.981	64
8	1.357	64
16	1.472	256

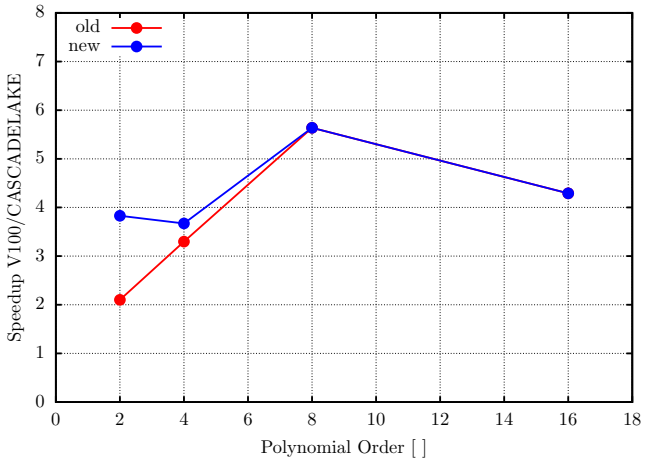


Figure 3: Table and graph show optimization impact on performance, thread-block size and speed-up 1 V100 GPU vs 1 Cascade CPU node. The table shows that the block size for the optimized code increased for orders n=2,4.

The table and chart in Figure 3 show the performance impact of the optimizations. The speed-up of the new version outperforms an Intel Cascade Lake 40-core CPU by a factor of 4–5.5X depending upon order.

Radiative Heating Simulation Code Takes OpenACC Optimizations into Production

NASA relies on aerothermodynamic simulation codes to define the convective and radiative heat transfer environments experienced by our planetary entry vehicles (Artemis/Orion, Mars 2020, Dragonfly, etc.). These environment predictions are critical to the design of the materials used for each vehicle's Thermal Protection System (TPS), as well as the layout and thickness of the selected materials. As such, rapid and accurate aerothermal simulation is a cornerstone of NASA's technical capability in the area of planetary Entry, Descent, and Landing (EDL). NEQAIR is one of two tools used by the agency to predict non-equilibrium radiative heat transfer. This is achieved by first computing the populations of electronically excited gas species in the flow field and calculating the radiation spectra generated, followed by solving the radiative transport equation to the vehicle surface. The NEQAIR logo in Figure 4 shows a simulated spectra (left) and an aeroshell flow-field

Figure 4: NEQAIR Logo

calculation (right).

NEQAIR performs computations on a two dimensional domain discretized in space and radiative energy (frequency/wavelength) with typical problem sizes of O($10^{\wedge 2}$x $10^{\wedge 6}$). The majority of the computational time is spent on the first part of the calculation (spectra generation), which can be solved independently across the smaller of the two dimensions and is parallelized by MPI. Within this calculation are several computations over the larger second dimension which are targeted to be offloaded to the GPU using OpenACC.

For the NASA GPU Hackathon, two mini-apps were created to quantify the speedup found when NEQAIR is accelerated using GPU hardware and targeted the main computational bottlenecks of the code: a *spectral broadening* function (essentially a convolution) and the *bound-free radiation* (an interpolation followed by algebraic manipulation of large arrays). These two routines account for roughly two-thirds of the NEQAIR runtime. Once those two routines were optimized, the next most time consuming routine identified was for the *broadening of atomic lines*.

The team started out with an OpenACC implementation developed during a hackathon in 2017. The 2017 implementation was slower than CPU-only code. During the current NASA GPU Hackathon the mini-apps were able to be sped up ~10-30x depending on the case on a V100 node using 2 or 4 GPUs. The team successively added OpenACC *"parallel loop"* directives to the time-consuming routines listed above. The two main breakthroughs for the optimizing the molecular broadening routine were to firstly reverse the loop order within the offloaded kernels, which avoided using atomics, and secondly, to check for and skip calculations on zero valued entries. A key component for porting the mini-apps to the main code was figuring out how to run MPS (Multi-Process Server) since we are in an MPI environment. MPS allows multiple CUDA processes to share a single GPU context. The partitioning of the GPUs to MPI ranks was achieved by using *MPI_COMM_SPLIT_TYPE* and a sequence of runtime OpenACC runtime commands. Attention was also given to data movement and tracking what is on the device and host at various points throughout the calculation.

The NEQAIR mini-app updates were included into the main development branch of NEQAIR. The chart in Figure 5 shows the incremental performance improvement resulting from the optimizations. For performance measurement purposes, the ratio of cores/ranks per GPU was fixed at 9:1 when more than 9 ranks were employed. The performance was measured in terms of speed-up relative to a single core with no GPU. The updated code with molecular broadening and bound-free + molecular broadening offload was about 40% and 80% faster than the CPU code, respectively (on 9 ranks). Offloading the Atomic broadening routine, the speed up was increased to 4-5x. Additional offloading increased the speed up further. On 1-2 ranks, a speed up of ~8x is now realized. This speed up should enable improved through-put for 3D and coupled radiation calculations for future NASA mission vehicle design.

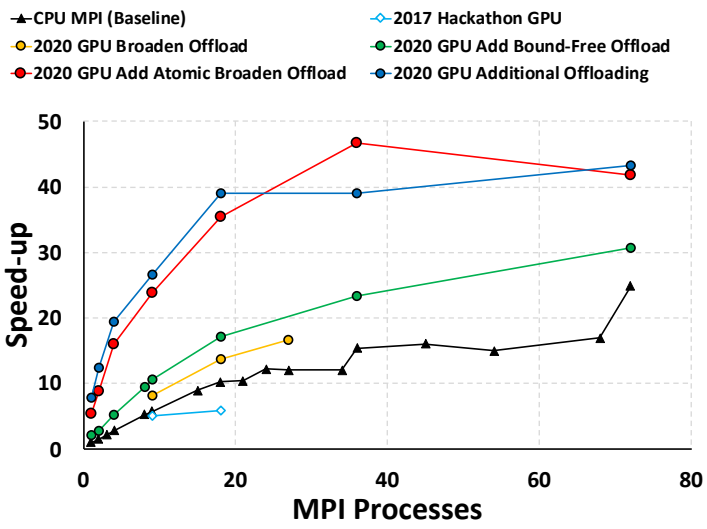


Figure 5: NEQAIR when successively off-loading more kernels to the GPU performance.

Accelerating Design of NASA's Planetary Entry Vehicles using Kokkos

Whenever NASA brings astronauts back from Earth orbit or sends a probe to Mars, a specially designed entry vehicle must be used to protect our precious cargo from the extreme heating generated when entering an atmosphere at over 10,000 miles per hour. For decades, NASA relied on LAURA and DPLR to predict the convective heat flux experienced by our entry vehicles. LAURA and DPLR are multi-block-structured, finite-volume CFD codes that solve the Navier-Stokes equations with extensions to accommodate the thermodynamic and chemical non-equilibrium effects that become important at planetary entry velocities. Both codes run on conventional CPU architectures and use the Message Passing Interface (MPI) standard to enable domain decomposition parallelization. The image in Figure 6 shows heating contours for the Space Shuttle during re-entry, as computed with DPLR.

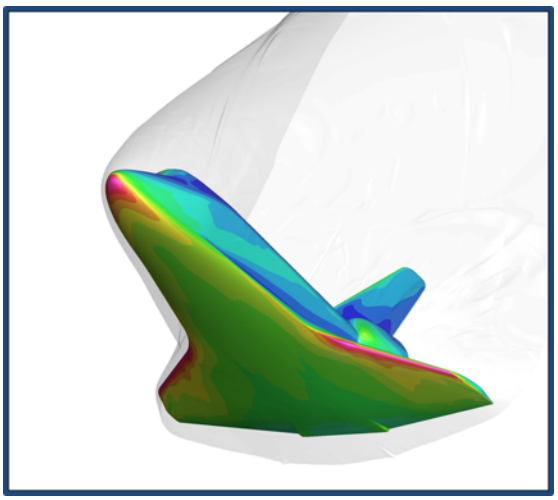


Figure 6: Surface heating contours for the Space Shuttle during re-entry, as computed with DPLR.

For the NASA GPU Hackathon, the LAURA and DPLR development team NFLOW-CHEM brought forward a pair of mini-apps representative of key computational kernels in our codes. The first mini-app developed is representative of the Symmetric Total-Variation-Diminishing (STVD) flux scheme used in LAURA. The primary challenge for this mini-app is managing spatial and temporal data locality, as the STVD scheme involves data access from a 13-point stencil at each cell in the structured grid. The second mini-app is derived from DPLR, and evaluates and linearizes the high temperature, finite-rate chemical kinetic models utilized in both codes. This mini-app is an embarrassingly parallel loop over the cells of a computational grid, computing the net species production and non-equilibrium heat release in each cell due to an Arrhenius-style 19-reaction air chemistry model suitable for Lunar return re-entry analysis.

Both mini-apps utilized the Kokkos Performance Portability Library as the programming model for GPU porting. This library was selected because it provides a highly customizable parallel execution framework in standard C++ that enables controlling data layout and performing computation in parallel using CPUs and/or GPUs from a variety of hardware vendors.

```
Kokkos::parallel_for(
    "Initialize State",
    Kokkos::MDRangePolicy<ExecSpace, Kokkos::Rank<3>>{{0,0,0}, {nx,ny,nz}},
    KOKKOS_LAMBDA (const size_t i, const size_t j, const size_t k){
        const auto f = get_flow_state(i, j, k, nx, ny, nz);
        size_t e = 0;
        for (auto d : f.densities)    state(i,j,k,e++) = d;
```

Figure 7: A simple example of dispatching a three-level tightly nested loop for parallel execution in Kokkos. This code can run as written on CPU and GPU depending on the "ExecSpace" (Execution Space) selected at compile time. Kokkos will alter the data layout of "state" and the i/j/k iteration order of the loop to promote cache-friendly access on CPU and coalesced memory access on GPU.

Specifically, Kokkos provides a multidimensional array type that can be specialized for a variety of memory layouts, including a tiled memory layouts designed for stencil operations. Selection of the memory layout can be changed at compile time without changing any of the code that actually uses the array. Kokkos also offers mechanisms for expressing hierarchical parallelism, which allows for precise control of how work is parallelized on CPU and GPU. Figure 7 shows a Kokkos code example.

At the end of the hackathon, the mini-app performance was measured for a variety of problem sizes on both a conventional dual-socket Intel Xeon Skylake compute node and a single NVIDIA V100 GPU. Figure 8 shows that utilizing the V100 GPU with the Kokkos CUDA backend the team achieved ~11.5x for the Flux mini-app for an 11 species reacting air model typically used for analysis of lunar return trajectories. The Kinetics mini-app achieved a ~5x speed-up. In particular, Kokkos' ability to rapidly explore different threading strategies for the GPU was crucial for maximizing performance and strong scaling of the DPLR chemical kinetics mini-app. The Kokkos tiled memory layout was also found to be very performant on both GPU and CPU architectures when utilized by the LAURA inviscid flux mini-app.

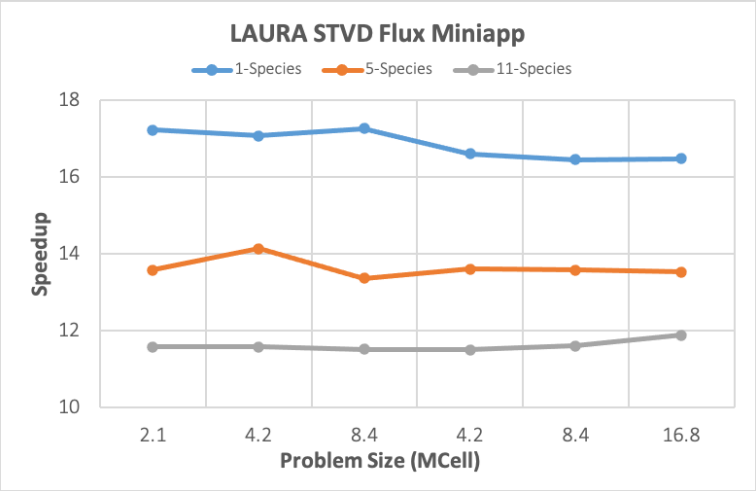


Figure 8: Speedup in execution time using a single NVIDIA V100 GPU relative to using all 40 cores of a dual-socket Intel Xeon Skylake node.

Heat Transfer in Turbomachinery Jump-starts on GPU with OpenACC

Glenn-HT is a CFD code that is in continuous use and development at NASA Glenn Research Center. It is extensively used for calculations of heat transfer in turbomachinery. It is implemented in Fortran 90. Parallelization is based on domain decomposition and MPI. Numerical algorithms employed are explicit Runge-Kutta methods, implicit alternating-direction solvers and red-black Gauss-Seidel solvers for linear systems. In addition, multigrid algorithms are used for convergence acceleration for non-linear partial differential equations. The image in Figure 9 shows the result of an 85 million mesh simulation using the full application.

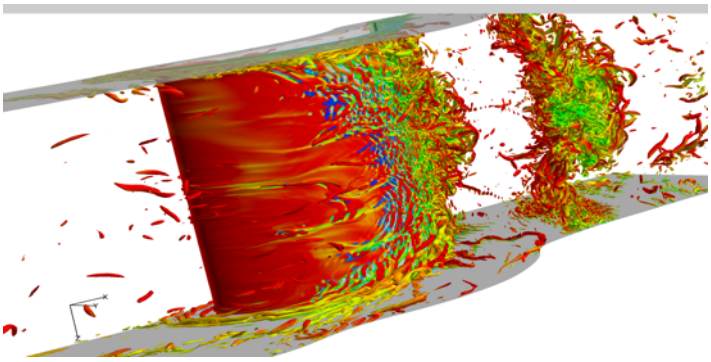
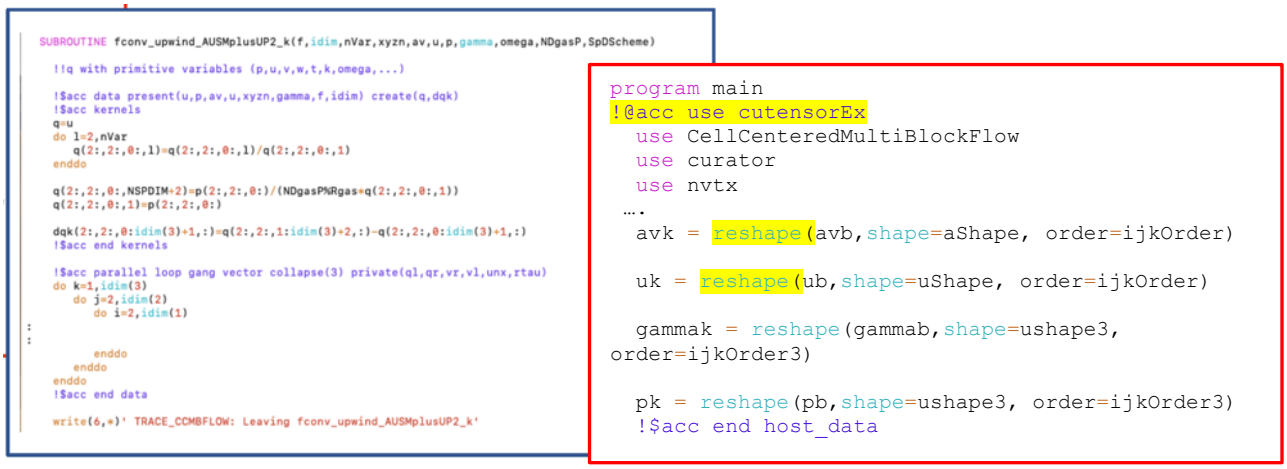


Figure 9: Iso-surface of Q-criterion colored by normalized total pressure around the Low-Pressure Turbine blade.

The goal of Team Glenn-HT was to develop a strategy for using GPU to accelerate computing and enable fast unsteady and LES simulations for multi-blade row compressors and

turbomachinery flows. The focus of the Hackathon was acceleration of the AUSM (Advection Upstream Splitting Method). The challenge when employing this scheme is, that for each surface (~$10^{\wedge 6}$-$10^{\wedge 7}$ surface), based on the gradient of each of the variables, the *flux limiter*, i.e. the Monotonic Upstream-centered scheme, has to be evaluated to avoid numerical instability. Another challenge of the application is the frequent call to a reshaping routine, which does not perform floating point calculations, but merely memory copies.
In order to execute the time consuming compute loop on the GPU, the team employed the OpenACC *"kernels"* and "*parallel loop*" directives. They found that using "*parallel loop gang vector collapse(3)*" significantly reduces the computational time of the main loop. Minimizing the communication time for the data exchange between CPU and GPU was achieved by employing the *"create"* and *"present"* clauses where appropriate. In order to address the performance of reshaping the arrays, the team employed the Nvidia *cutensorEx,* which significantly reduced the execution time. The code changes are shown in Figure 10. Overall, Glenn-HT reported a speed-up of almost 20X when running the mini-app on the accelerator. They are now working on a CUDA implementation. Profile in Figure 11shows the incremental performance improvement.



```
SUBROUTINE fconv_upwind_AUSMplusUP2_k(f,idim,nVar,xyzn,av,u,p,gamma,omega,NDgasP,SpDScheme)

  !!q with primitive variables (p,u,v,w,t,k,omega,...)

  !$acc data present(u,p,av,u,xyzn,gamma,f,idim) create(q,dqk)
  !$acc kernels
  q=u
  do l=2,nVar
     q(2:,2:,0:,l)=q(2:,2:,0:,l)/q(2:,2:,0:,1)
  enddo

  q(2:,2:,0:,NSPDIM+2)=p(2:,2:,0:)/(NDgasP%Rgas*q(2:,2:,0:,1))
  q(2:,2:,0:,1)=p(2:,2:,0:)

  dqk(2:,2:,0:idim(3)+1,:)=q(2:,2:,1:idim(3)+2,:)-q(2:,2:,0:idim(3)+1,:)
  !$acc end kernels

  !$acc parallel loop gang vector collapse(3) private(ql,qr,vr,vl,unx,rtau)
  do k=1,idim(3)
     do j=2,idim(2)
        do i=2,idim(1)
:
:
        enddo
     enddo
  enddo
  !$acc end data

  write(6,*)' TRACE_CCMBFLOW: Leaving fconv_upwind_AUSMplusUP2_k'
```

```
program main
!@acc use cutensorEx
  use CellCenteredMultiBlockFlow
  use curator
  use nvtx
 ….
  avk = reshape(avb,shape=aShape, order=ijkOrder)

  uk = reshape(ub,shape=uShape, order=ijkOrder)

  gammak = reshape(gammab,shape=ushape3,
order=ijkOrder3)

  pk = reshape(pb,shape=ushape3, order=ijkOrder3)
  !$acc end host_data
```

Figure 10: Source code sketching routines AUSM and Reshape. The left shows how *present* and *create* clauses were employed, the code on the right shows the use of *the cutensorEX.*

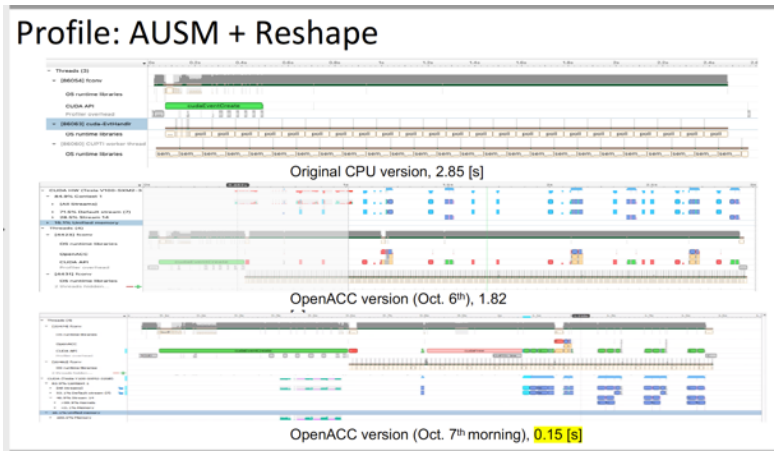


Figure 11: Nvidia nsys profiles for different implementations of routines AUSM and Reshape.

Restructuring and Replacing OpenACC with CUDA Gives Performance Boost to LAVA Lattice Boltzmann Mini-App

NASA's LAVA team created a Lattice Boltzmann Method mini-app to understand how to port their full app to GPU. The full app is based on block-structured Cartesian adaptive mesh refinement (AMR) with immersed boundaries, and is used for low-speed aerodynamics, as can be seen in Figure 12. The mini-app is a significantly simplified version of the full app and is based on a uniform-refinement with block partitioning, excluding complex geometry.

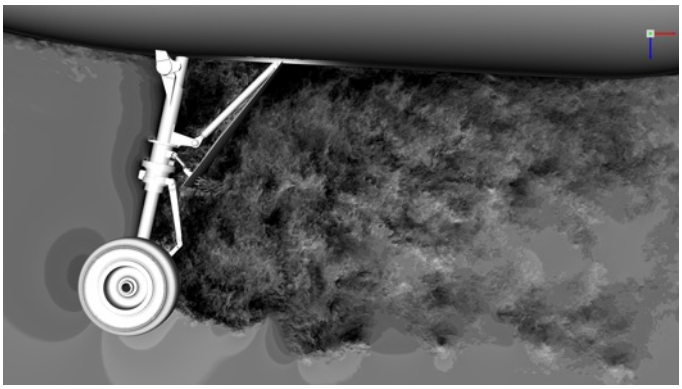


Figure 12: Simulation using Intel CPU version of LAVA Lattice Boltzmann of nose landing gear, where colors indicate velocity magnitude (white is fast , black is slow)

The mini-app used for the NAS Hackathon ran on a single level of uniform, isotropic refinement. Its performance is limited by memory bandwidth. The performance of the benchmark is measured in millions of cell updates per second (MUPS). Team LAVA started from an earlier implementation of the mini-app which was used during a 2018 hackathon and was ported to the GPU using OpenACC directives. It achieved a performance of 214 MUPS on V100 GPU nodes at the 2018 hackathon, where the implementation was restricted to a single 256^3 box. For the 2020 Hackathon, this was updated with a massive partitioning of the same 256^3 domain into 8^3 boxes. This is work towards block-structured AMR where more than O(10^5) boxes are typically used. For the 2020 Hackathon, the team focused on optimization to perform iterations over the many small boxes efficiently on the GPU. The code consists of two main steps: stream and collide. The streaming step is composed of no floating point operations and uses halos for memory copies (27-point stencil) with accompanying integer arithmetic. The collide step is a massive 1D loop over all lattice sites where O(100) doubles per lattice site are combined with floating point operations.

Git Hash	MUPS	Details
787b18fc	214	Code from 2018 Hackathon (230 was obtained in 2018...hardware?)
4b4dc29d	0.24	Multi-Box Implementation (broken acc pragmas)
b639874c	57.1	Fixed acc pragmas
c6308551	83.8	Chunking implemented
af1bcd4b	87.1	Gang/Vector looping when getting halo data
24b34c67	113	Improved chunking
52817003	170	Fused halo exchange and streaming
b4134635	177	Stopped storing "gamma", computing locally
e8403b2b	180	Simplified index computation in collide
cea6c09b	182	Switched to 1d memory storage (not clear if this is worth it)
2be712bc	189	Fused computeMacroscopic() and collide()
659bffe2	233	1D Looping in collide
b13a9d5c	284	Switched to CUDA for Stream (utilizing 1D looping, and redundant index calcs)
40e0c7a7	363	Switched to CUDA for Collide (with 1D looping)
-	430	Turned off integrate statistics at every step [no longer apples-apples]

Figure 13: Incremental performance improvements of NASA-LAVA on a V100 node.

As a first step, the 2018 OpenACC implementation had to be adjusted to allow for multi-box execution. Then the team successively implemented a number of optimizations. For example, adding efficient chunking (i.e. fusing of small boxes into larger work units) and employing gang/vector looping when getting halo data yielded a performance improvement 2X. One large kernel was launched which simultaneously spans all the boxes and assigned one gang (OpenACC) or thread block (CUDA) to each box. For the collide step the loop over the boxes was collapsed such that there was no need for hierarchical parallelism. Another nice performance

increase resulted from fused halo exchange and streaming. Previously the code performed the halo exchange to a temporary array, which is where non-contiguous memory accesses occurred. Then the halo data was read into the corresponding locations in the box interior for the stream operation. This was combined into a single operation and read directly from the neighboring boxes, still paying the non-coalesced access penalty but only doing a single memory read.

The code is kept in a git repository, which allows the developers to keep track of the changes and their impacts on performance, as can be seen for a subset of commits in Figure 13. Most striking is the performance boost when switching to the use of CUDA for some of the routines. Other findings were that multidimensional vector<vector<double>> performed poorly. Switching to single vector<double> resolved this issue. The team anticipates even further speed-ups for A100 due to further improved memory bandwidth.

NASA GPU Hackathon Yields Significant Code Improvements

HECC and Nvidia jointly organized the NASA GPU Hackathon 2020 to bring together application developers and computer experts to help get important NASA applications running effectively on graphics processing unit (GPU) nodes. Nine teams of application developer participated in this event , a major impetus for teams to modernize codes of interest for NASA missions to CPU nodes containing GPU accelerators, with a focus on hands-on problem solving. HECC provided five Pleaides nodes with V100 GPUs for teams to use during the event.

During the event, which focused on AeroSciences and CFD applications, most teams achieved considerable performance improvements on both GPUs and CPUs. For example, a team with no GPU experience completed a first port of a time-critical loop to a GPU.
Another team of expert CUDA programmers were able to restructure their algorithm, yielding a factor of five speed-up. And another team sped up some of their CUDA kernels by a factor of 20, which directly translated into their production code.