

Performance Portability Issues for a Large-Scale Computational Fluid Dynamics Application on Emerging High-Performance Architectures

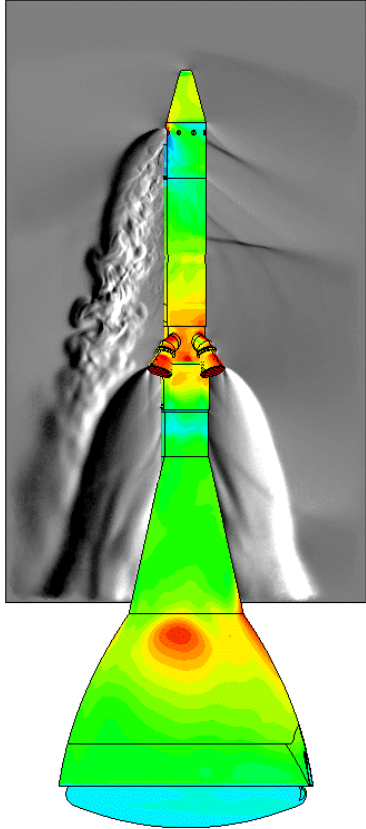
Performance, Portability, and Productivity in HPC Workshop
September 1-2, 2020
(Originally April 7-9 in Kansas City, MO)

Aaron Walden
NASA Langley Research Center
Hampton Virginia

Mohammad Zubair
Old Dominion University
Norfolk Virginia

Eric Nielsen
NASA Langley Research Center
Hampton Virginia

Applications

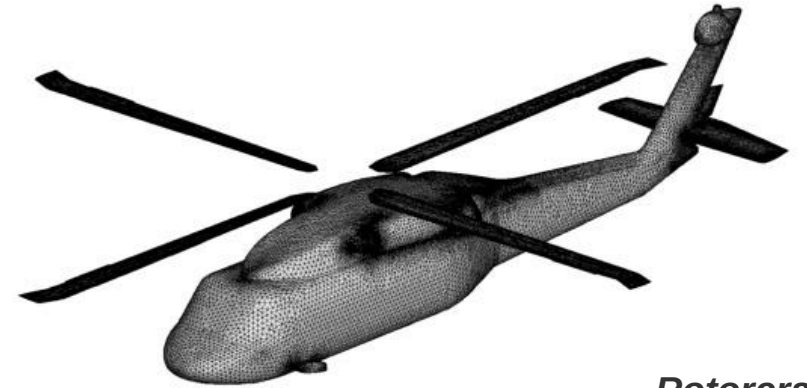


**Launch Abort
System**

**Adjoint for
Chaotic
Systems
(NASA/MIT)**

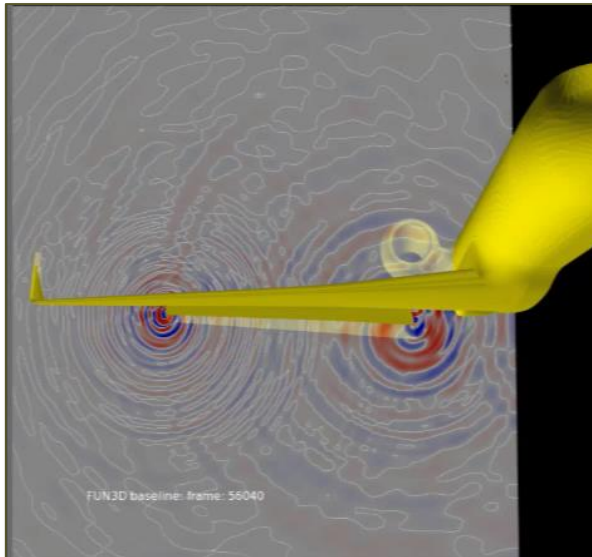


Time Step = 2000



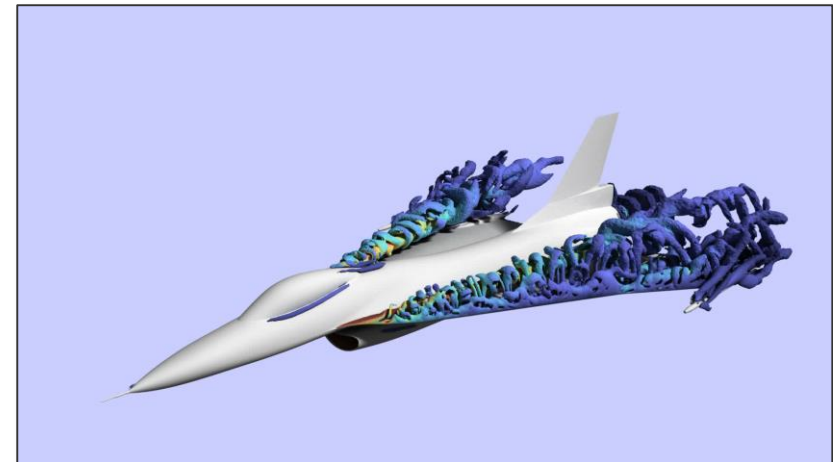
Rotorcraft

**NASA/Boeing
Truss-Braced Wing**



**Aeroacoustics:
Gulfstream
G550**

**Separated
Flows**



Performance on Emerging Architectures

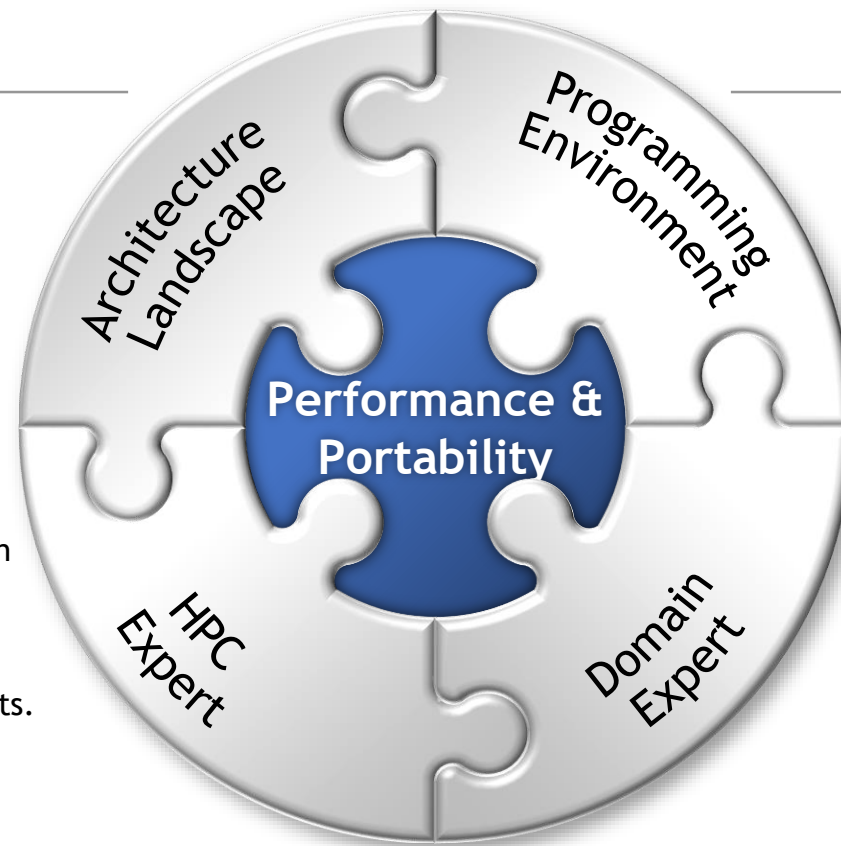
Enable science that was not possible earlier in a cost-effective way!

Architecture

- GPU: NVIDIA, AMD
- CPU: ThunderX2, Fujitsu
- VECTOR: NEC Vector Engine

HPC Expert

- Works closely with domain expert
- Understand computer architecture and programming environments.



Programming Environment

- NATIVE: Fortran/C/C++, CUDA, HIP
- PORTABLE: OPENCL, SYCL, OCCA, IntelOneAPI, Kokkos, Raja, ..
- Works closely with HPC expert
- Enable modifications to the computations to match the underlying architecture

Domain Expert

- Works closely with HPC expert
- Enable modifications to the computations to match the underlying architecture

Porting a Large Application to Diverse Architectures

- Many cores CPU (e.g. Intel Skylake, ARM ThunderX2)
 - Parallelization across many cores
 - Memory and cache utilization
 - Vectorization
 - NUMA issues
- Accelerator: GPU (e.g. NVIDIA Tesla V100/A100 and AMD MI25/60)
 - Accelerator Issue (Heterogeneous computing)
 - Fine grain data parallelism
 - Memory and cache utilization
 - SIMT architecture (Warp/Subgroup)

Requires offloading almost the full application before you see performance benefit

Accelerator Issues

Algorithm 1 Dummy Application

while *not converge* **do**

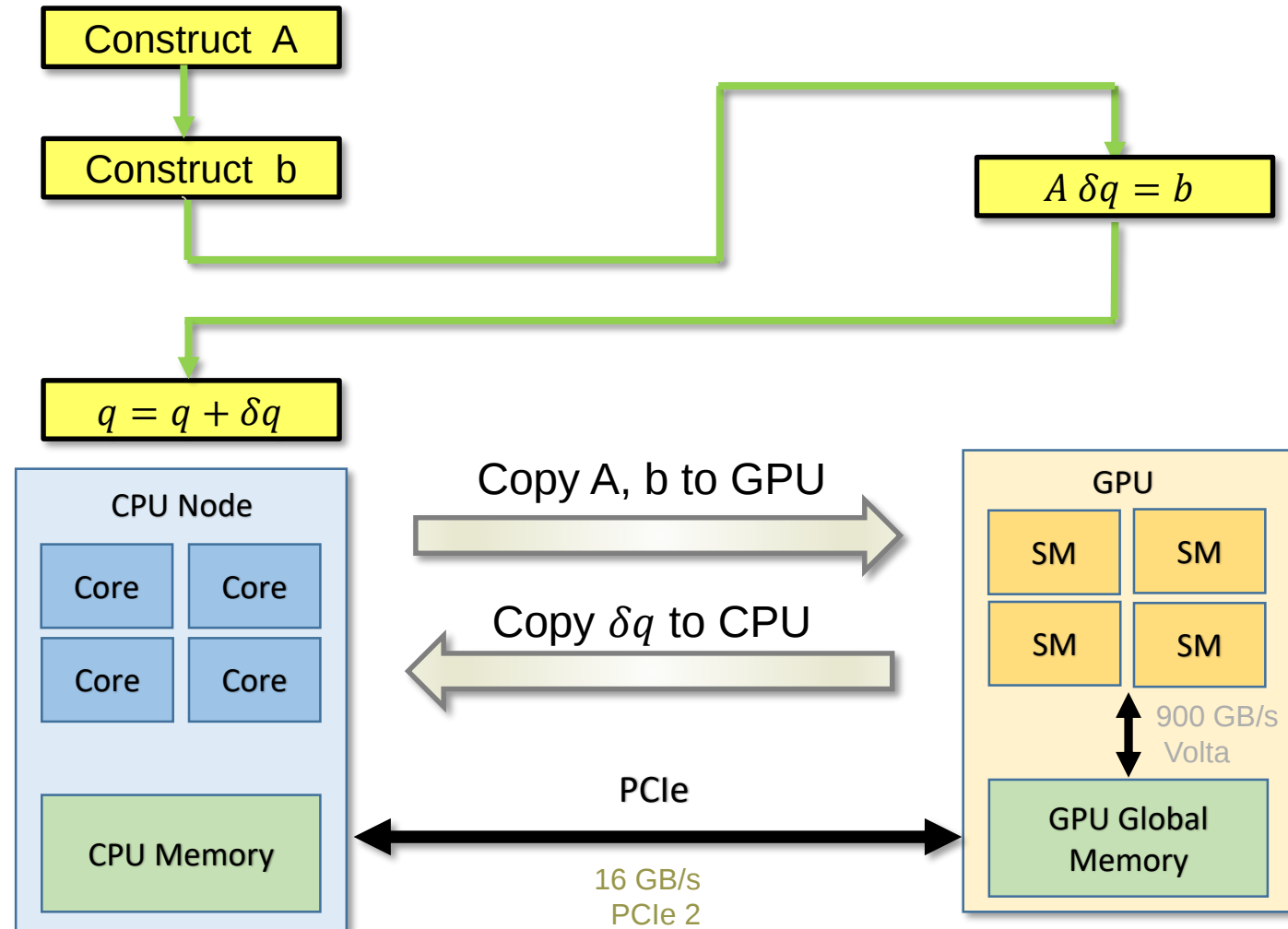
 Construct A

 Construct b

 Solve $A\delta q = b$

 Update $q = q + \delta q$

end



Accelerator Issues

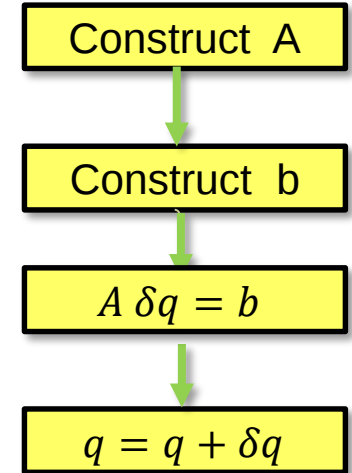
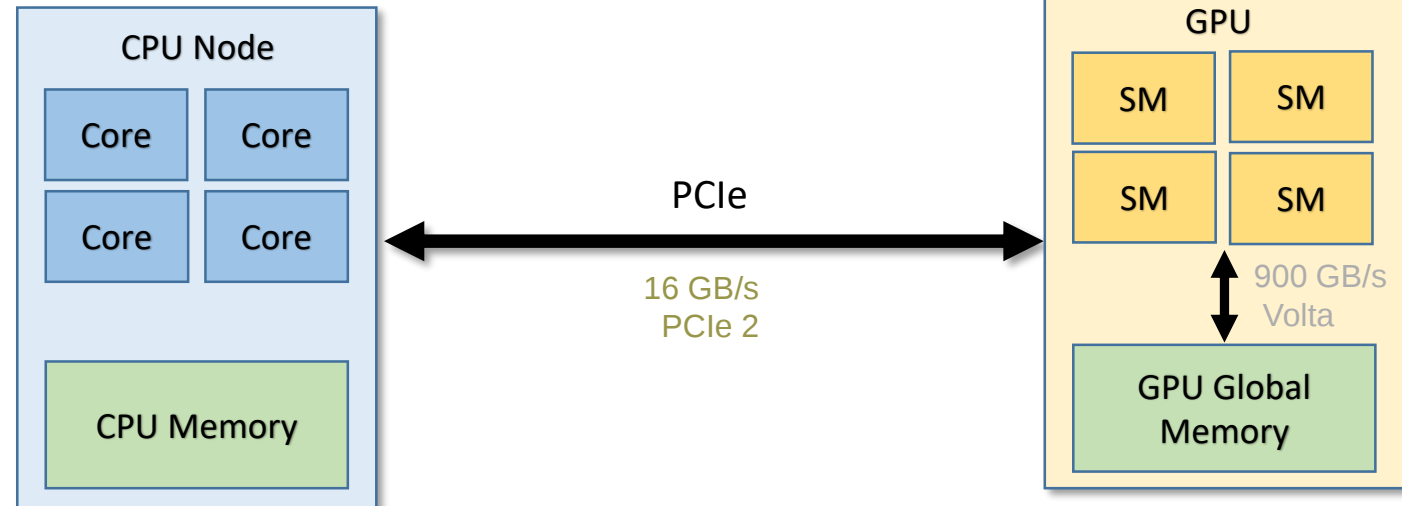
Algorithm 1 Dummy Application

while *not converge* do

Construct A
Construct b
Solve $A\delta q = b$
Update $q = q + \delta q$

end

Move all four kernels to GPU to avoid frequent movement of data between CPU and GPU



Two Dimensions of Performance Portability

- Target HPC Architecture
 - Multicore CPU: Intel Xeon, ARM ThunderX2, ARM A64FX, NEC Vector Engine (large vector)
 - GPU: NVIDIA V100/A100, AMD MI 60/MI100, Intel Xe
- Programming Environment
 - Support CPU and GPU: OpenACC, OpenMP, SYCL, OCCA, Kokkos, RAJA, etc.
 - A single code base that supports both CPU and GPU. The single code base may query the underlying system and take different paths in the code depending on the underlying architecture (e.g. SYCL)
 - Support GPU only: CUDA, HIP

Questions:

Can we maintain optimal performance on a given architecture when we code a computation using a different programming environment?

Can we maintain optimal performance on different architectures when we use a single code base written using a programming environment that supports diverse architectures (e.g. SYCL)?

- FUN3D solves the Navier-Stokes fluid dynamics equations using implicit time integration on general unstructured grids
- An approximate nearest-neighbor linearization of the residual equations for each control volume gives rise to a large tightly-coupled system of block-sparse linear equations

Algorithm 1 SOLVER

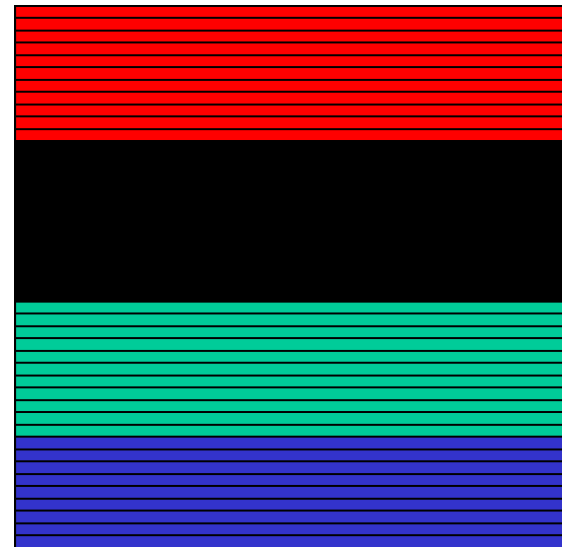
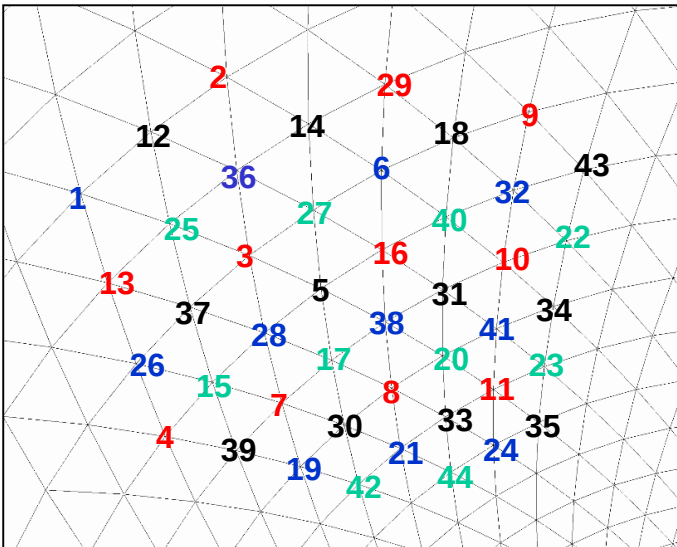
```
1:  $q \leftarrow 0$ 
2: for  $i = 1$  to maxiter do
3:   Construct Jacobian matrix  $A$  at  $q$ 
4:   Construct vector  $b$  at  $q$ 
5:   Solve for  $\Delta q$  in linear system  $A\Delta q = b$ 
6:    $q \leftarrow q + \Delta q$ 
7: end for
```

There is no communication across nodes as the code can be parallelized over number of cells with atomic updates. A large number of variables and temp arrays results in registers and cache pressure on GPU device.

This requires communication across nodes and is critical for the scalability. Focus of the talk will be on this kernel.

FUN3D Multicolor Linear Solver

- Color by rows which share no adjacent unknowns; and re-order matrix rows by color contiguously
- Unknowns of the same color carry no data dependency and may be updated in parallel
- Updates of unknowns of each color use the latest updated values at grid points corresponding to other colors
- The overall process may be repeated using several outer sweeps over the entire system



FUN3D Multicolor Linear Solver

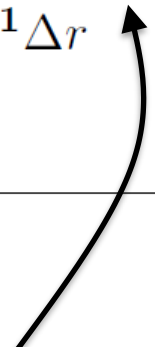
- Implicit scheme results in linear systems of equations:
 - $A \Delta q = b$, A is a sparse $n \times n$ block matrix for n grid points; block is of size $nb \times nb$
- Matrix A is segregated into two separate matrices:
 - $A \equiv O + D$, where O and D represent the off-diagonal and diagonal blocks of A
- Prior to performing each linear solve, each diagonal block D is decomposed in-place into lower and upper triangular matrices

- Solver Challenges
 - With an average of few off-diagonal blocks per row, the arithmetic intensity of the computation is quite low (≈ 0.5) – memory bound on GPU
 - The number of rows associated with a color can vary significantly. Consequently the amount of parallelism available for different colors varies significantly

Algorithm 2 LINEAR SOLVER

```
1: for  $i = 1$  to  $niter$  do
2:   for  $c = 1$  to  $nc$  do
3:      $\Delta r = b_c - O_c \Delta q$ 
4:      $\Delta q = D_c^{-1} \Delta r$ 
5:   end for
6: end for
```

Block-sparse matrix-
vector product



Optimization Issues for Multicolor Linear Solver

- To understand performance portability issues associated with a particular kernel, it is critical to understand the optimal performance that may be achieved on each target architecture.
 - This typically requires a specific implementation targeting each individual architecture, leveraging low-level approaches that are not portable.
 - Optimized implementations of the linear solver kernel have been established for all of the target platforms considered

To achieve optimal performance on the Intel Xeon Skylake and Marvell ThunderX2 CPUs, kernels were developed using AVX-512 and NEON vector intrinsics, respectively.

An optimized CUDA implementation is used for the NVIDIA GPU architecture.

To develop an optimal implementation for the AMD GPU, the CUDA kernel has been transformed to a HIP implementation with specific customizations targeting the AMD architecture.

Several higher-level programming models are currently being explored to provide a single-source approach capable of achieving reasonable performance across architectures.

Optimization on Intel Skylake

ARCHITECTURE	CRITICAL PERFORMANCE ISSUES	IMPLEMENTATION	OPTIMIZATION EFFORT	% OF PEAK MEMORY BANDWIDTH	TIME
Intel Skylake 6148 (Dual Socket)	• NUMA: minimize memory traffic between two NUMA nodes • Vectorization: utilization of the Intel vector unit based on AVX-512 • Cache and memory utilization	• Use of hybrid implementation, MPI + OpenMP, with an MPI rank on each NUMA node and OpenMP threads for cores within a node • For vectorization, map a complete non-zero block to two 512-bit vectors • Use of vector loads and cache prefetching • Language: C and Intel Vector Intrinsics	Hard	84%	11 ms

Optimization on ARM ThunderX2

ARCHITECTURE	CRITICAL PERFORMANCE ISSUES	IMPLEMENTATION	OPTIMIZATION EFFORT	% OF PEAK MEMORY BANDWIDTH	TIME
ARM Thunder X2 CN 9975 (Dual Socket)	• NUMA: minimize memory traffic between two NUMA nodes • Vectorization: utilization of the NEON vector unit • Cache and memory utilization²	• Use of Hybrid implementation, MPI + OpenMP, with an MPI rank on each NUMA node and OpenMP threads for cores within a node • For vectorization map partial columns of a non-zero block to a 128-bit vector • Use of vector loads and cache prefetching • Language: C and NEON Vector Intrinsics²	Moderate	62%	11 ms

Optimization on NVIDIA Tesla V100 GPU

ARCHITECTURE	CRITICAL PERFORMANCE ISSUES	IMPLEMENTATION	OPTIMIZATION EFFORT	% OF PEAK MEMORY BANDWIDTH	TIME
NVIDIA Volta V100	• GPU global memory and shared memory utilization • Enable fine-grained parallelism for sparse computation • Minimize thread divergence - effective SIMT execution using warp of 32 threads • Use of advanced instructions that share data within a warp without going through memory	• Perform coalesced memory loads by mapping a warp of 32 threads to a non-zero block of 25 elements • A warp processes a row block and aggregate partial results from different threads using shuffle instruction • Use of shared memory to store intermediate result and also for storing elements of diagonal blocks • Language: C/C++ CUDA	Hard	73%	3.3 ms

Optimization on AMD MI50 GPU

ARCHITECTURE	CRITICAL PERFORMANCE ISSUES	IMPLEMENTATION	OPTIMIZATION EFFORT	% OF PEAK MEMORY BANDWIDTH	TIME
AMD Instinct MI50	• GPU global memory and shared memory utilization • Enable fine-grained parallelism for sparse computation • Minimize thread divergence - effective SIMT execution using wavefront of 64 threads • Use of advanced instructions that share data within a wavefront without going through memory	• Perform coalesced memory loads by mapping a wavefront of 32 threads to process two row-blocks with first 32 threads of the wavefront are mapped to a non-zero block of the first row, and the rest of the 32 threads are mapped to a non-zero block of the second row • Use of shared memory to store the intermediate result and also for storing elements of diagonal blocks • A 1/2 wavefront processes a row block and aggregate partial results from different threads using shuffle instruction • Language: HIP, C/C++ runtime API. HIP code is generated by transforming optimized CUDA code	Easy (starting from optimized CUDA code)	51%	4.3 ms

Performance Portability: Solver Kernel Using Different Programming Environments on V100

```
cgh.parallel_for<class solver_point5>(
    sycl::nd_range<2> {sycl::range<2>(gdimx, BLOCK_DIM_Y),
        sycl::range<2>(BLOCK_DIM_X, BLOCK_DIM_Y)},
    [=](sycl::nd_item<2> item) {

    int const tidx = item.get_local_id(0);
    int const tidy = item.get_local_id(1);
    int n = start + item.get_group(0) * BLOCK_DIM_Y + tidy - 1;
    int const k = tidx % 5;
    int const l = tidx / 5;
    float fk;
    double f1, f2, f3, f4, f5;
    int jam0, j;
    int istart = diam[n];
    int iend = diam[n + 1] - 1;

    if ( (n < end) && (l < 5)) {

        fk = 0;

        for (j = istart - 1; j < iend; j++) {
            jam0 = djam[j] - 1;
            fk += A_OFF(k, l, j) * DQ(1, jam0);
        }

        sm_f[k][l][tidy] = fk;
    }
    item.barrier(sycl::access::fence_space::local_space);
}
```

SYCL

CUDA

```
#pragma acc loop vector
for ( int tidx = 0; tidx < 32; tidx++) {
    int k = tidx % 5;
    int l = tidx / 5;
    if ((n < end) && (l < 5)) {
        int istart = iam[n] - 1;
        int iend = iam[n + 1] - 1;
        float fk = 0.0;
#pragma acc loop seq
#pragma unroll 2
        for ( j = istart; j < iend; j++) {
            icol = jam[j] - 1;
            fk += A_OFF(k, l, j) * DQ(1, icol);
        }
        sm_f[k][l][tidy] = fk;
    }
}
```

OPENACC

OCCA

```
if ( (n < end) && (l < 5)) {
    int istart = iam[n];
    int iend = iam[n + 1] - 1;

    float fk;
    double f1, f2, f3, f4, f5;

    fk = 0;
    int jam0, jam1;
    int j = istart - 1;

    for ( ; j < iend; j++) {
        jam0 = jam[j];
        fk += A_OFF(k, l, j) * DQ(1, jam0 - 1);
    }
}
```

```
for (int ty = 0; ty < NTY; ++ty; @inner(1)) {
    for (int tx = 0; tx < NTX; ++tx; @inner(0)) {

        int const k = tx % 5;
        int const l = tx / 5;
        int n = start + bx * NTY + ty - 1;

        if ( (n < end) && (l < 5)) {
            int istart = iam[n];
            int iend = iam[n + 1] - 1;

            double fk = 0.0;
            int jam0;
            int j = istart - 1;
#pragma unroll 4
            for ( ; j < iend; j++) {
                // jam0 = jam1;
                jam0 = jam[j];
                fk += A_OFF(k, l, j) * DQ(1, jam0 - 1);
            }
            sm_f[k][l][ty] = fk;
        }
    }
}
```

Portability vs Performance Portability

	Intel Skylake	ARM ThunderX2	NVIDIA V100	AMD MI-50	Intel PAC10 FPGA
OpenACC	TBD	TBD	1.1 Easy / Hard	TBD	No Plans
HIPSYCL	No Plans	No Plans	1.02 Moderate / Hard	TBD	No Plans
HIP	NA	NA	1 Moderate / Hard	1.32 Moderate / Hard	No Plans
SYCL For CUDA	NA	NA	1.02 Moderate / Hard	NA	No Plans
CUDA	NA	NA	1 Moderate / Hard	NA	No Plans
Vect. Intr. MPI	3.39 Easy / Hard	3.39 Easy / Hard	NA	NA	NA
OpenCL	TBD	TBD	TBD	TBD	200 Hard / Hard
OCCA	No Plans	No Plans	1 Moderate / Hard	1.32 Moderate / Hard	No Plans
Intel OneAPI	In Progress	No Plans	In Progress	In Progress	No Plans

- Numeric value indicates relative speed (lower is better)
- Ratings indicate effort for implementation / optimization
- **Red** indicates programming model that provides best performance for selected architecture

- Scientific kernels can greatly benefit from emerging high-performance architectures such as ARM, NVIDIA and AMD GPUs.
- For achieving performance on these architectures it is necessary to put effort in careful planning and optimization of computationally intensive kernels.
 - For optimizing code on a given architecture, it is necessary to understand the underlying architecture.
 - The general portability framework hides the architecture from the application developer and tries to generate optimized code for a given architecture. However, a number of applications require a restructuring of code which is difficult to do in an automated way or by a run-time environment.
 - For these applications, it is much easier to restructure the code at the application level to match the underlying architecture.
 - Assist vendors in developing optimized libraries of interest to NASA to ensure performance portability on newer models.