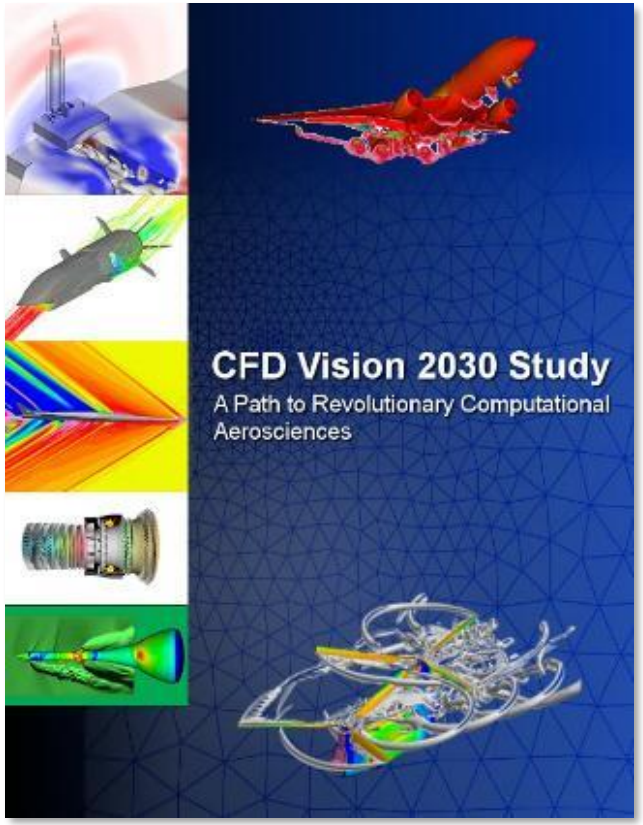
Experiences in Moving CUDA-Optimized Kernels to Intel GPUs using oneAPI

Mohammad Zubair — Old Dominion University
Chris Stone — National Institute of Aerospace
Aaron Walden — NASA Langley Research Center
Eric Nielsen — NASA Langley Research Center

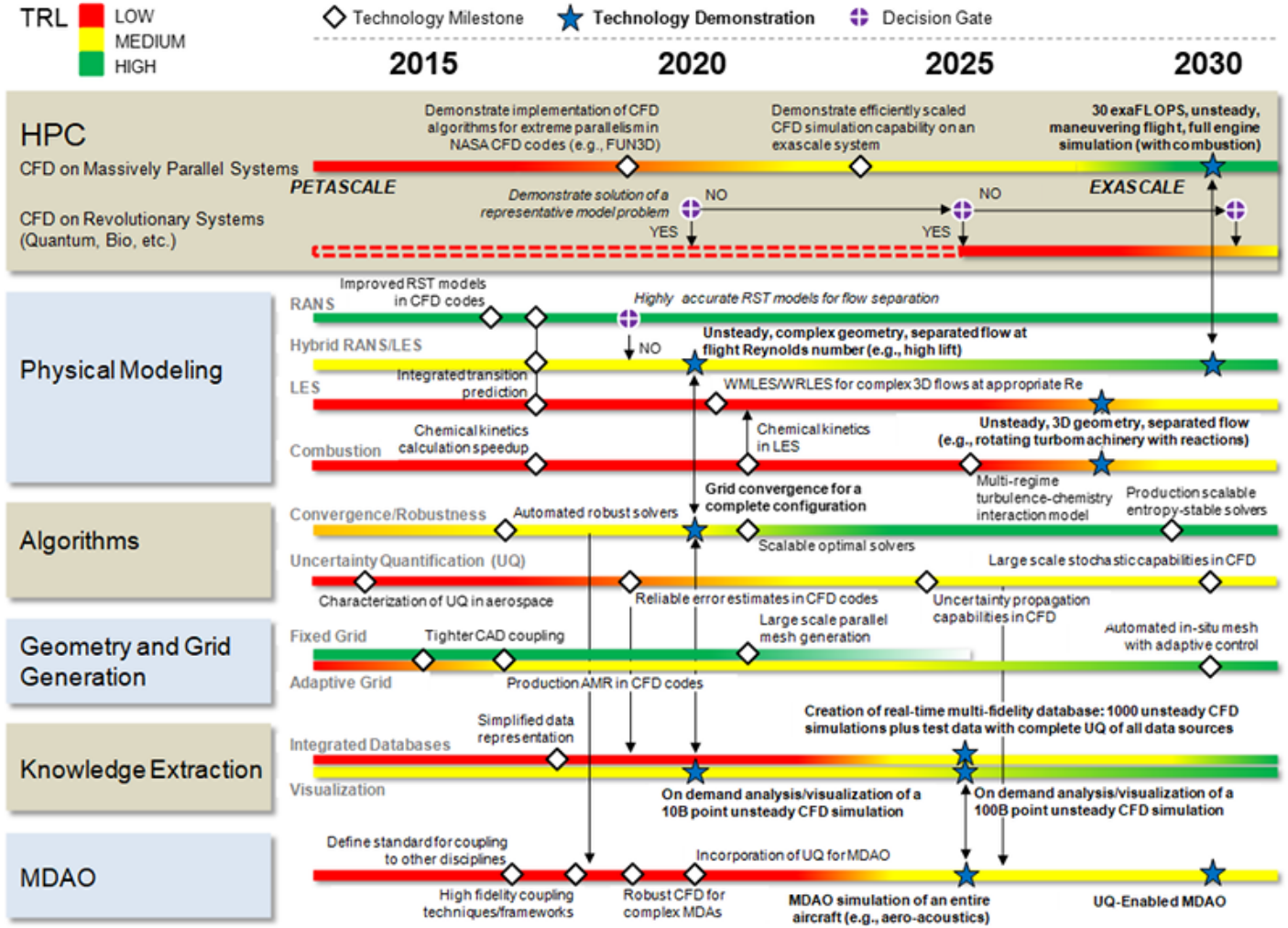
Outline

- Motivation
- Multicolor Point-Implicit Solver
- CUDA Approach for Optimization of Solver on GPU
- Porting CUDA Optimized Code on Pre-Production Intel GPU Hardware Using Intel oneAPI
  - Challenges and different approaches explored
- Conclusion

CFD Vision 2030 Study

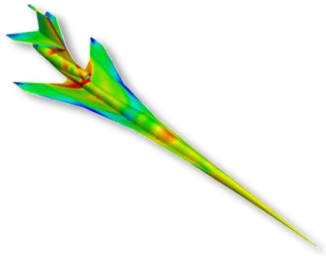


NASA/CR-2014-218178
See http://www.cfd2030.com

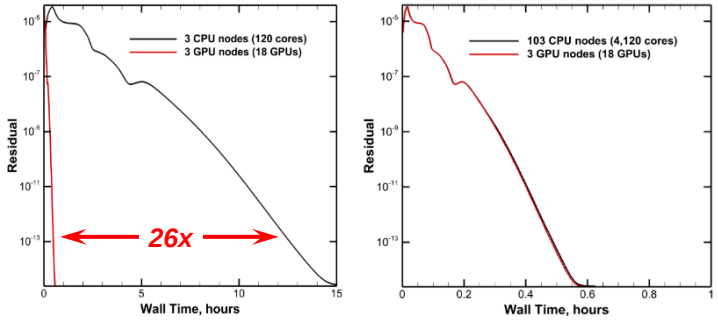


Examples of Engineering Use

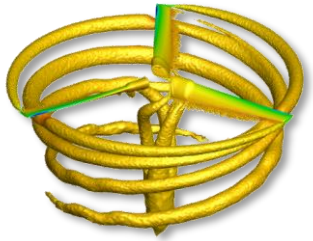
- NVIDIA V100 GPU improves over Intel Xeon Skylake CPU by 4-5x
  - NVIDIA A100 GPU improves to 7-8x
- GPUs typically bundled in nodes with 4, 6, or 8 GPUs
- GPU nodes are more expensive, but still a win on performance / $



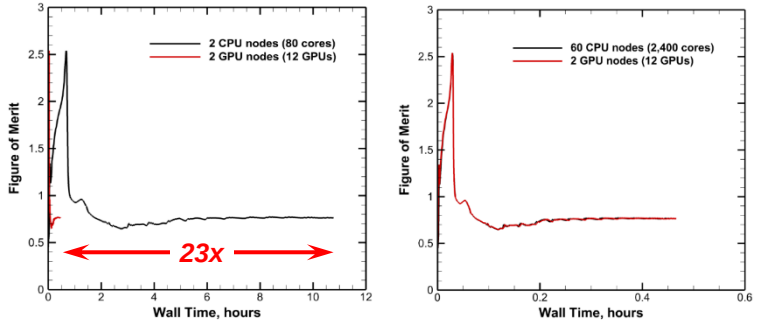
Supersonic Flows



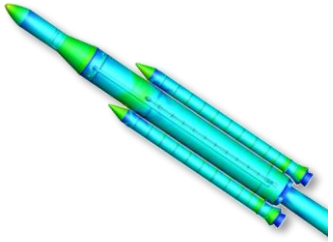
3 GPU nodes of 6xV100: 37 mins
3 CPU nodes of 120 cores: 16 hrs
OR
18 GPUs do the work of 103 CPUs (4,120 cores)



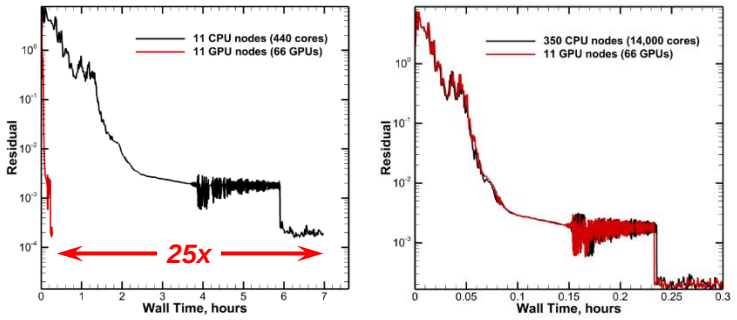
Rotorcraft



2 GPU nodes of 6xV100: 28 mins
2 CPU nodes of 80 cores: 11 hrs
OR
12 GPUs do the work of 60 CPUs (2,400 cores)



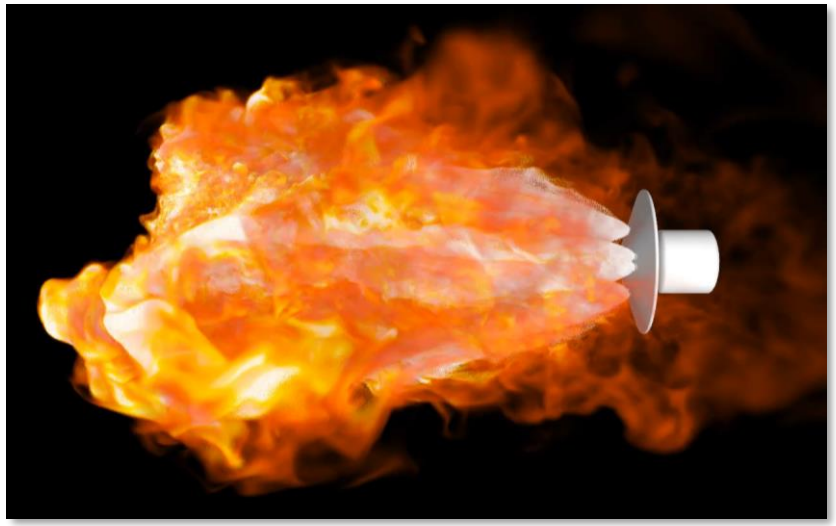
Space Launch System

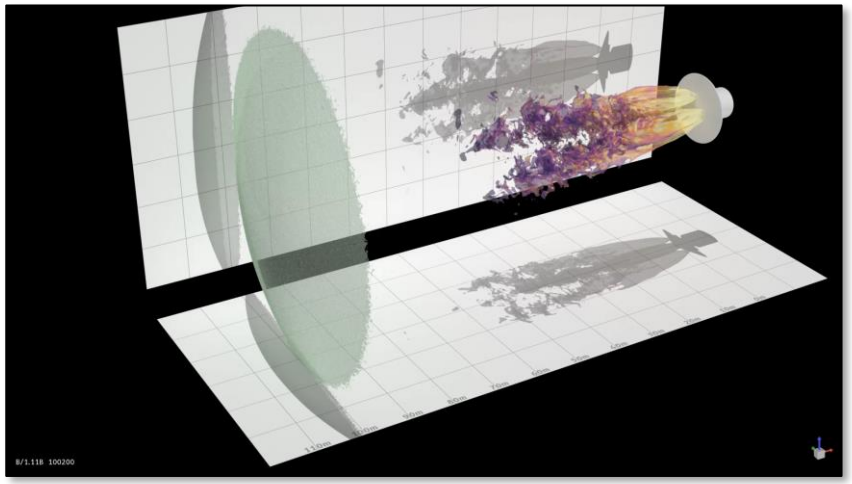


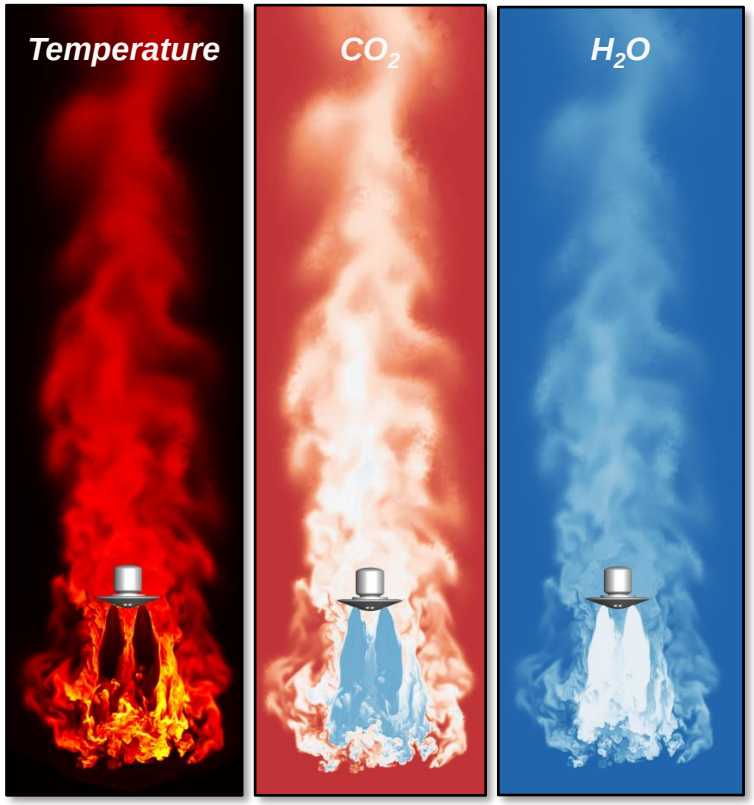
11 GPU nodes of 6xV100: 17 mins
11 CPU nodes of 440 cores: 7 hrs
OR
66 GPUs do the work of 350 CPUs (14,000 cores)

Recent Summit Campaigns

- Summit Early Science, INCITE campaigns for simulations of 16-meter diameter human-scale Mars lander with O_2/CH_4 combustion in Martian CO_2 atmosphere
- DES of 10 species/19 reactions, 7B elements, seconds of real time
- Runs on 15,912 V100s with approximate throughput of several million CPU cores
- Big data: 90 GB of asynchronous I/O every 30 seconds for 2 days yields ~1 PB per run; 60 TB/day migrated from ORNL to NASA Ames
- Stepping stone to CFD 2030 exascale milestones

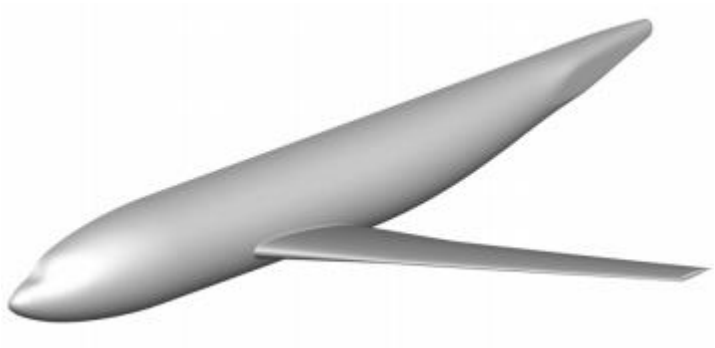






This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

FUN3D Test Problem



Transonic turbulent flow over a semi-span wingbody consisting of 1,123,718 grid vertices, 1,172,171 prisms, 3,039,656 tetrahedra, and 7,337 pyramids. The off-diagonal matrix consists of 19,106,474 blocks.

15 iterations of the linear solver are timed.

Multicolor Point-Implicit Solver

- FUN3D solves the Navier-Stokes equations of fluid dynamics using implicit time integration on general unstructured grids
- This approach gives rise to a large block-sparse system of linear equations that must be solved at each time step
- Multicolor point-implicit linear solver used to solve $A\,\Delta Q = R$

for i = 1 to n_time_steps
do
  Form Right Hand Side
  Form Left Hand Side
  Solve $A\,\Delta Q = R$
  Update Solution
end for

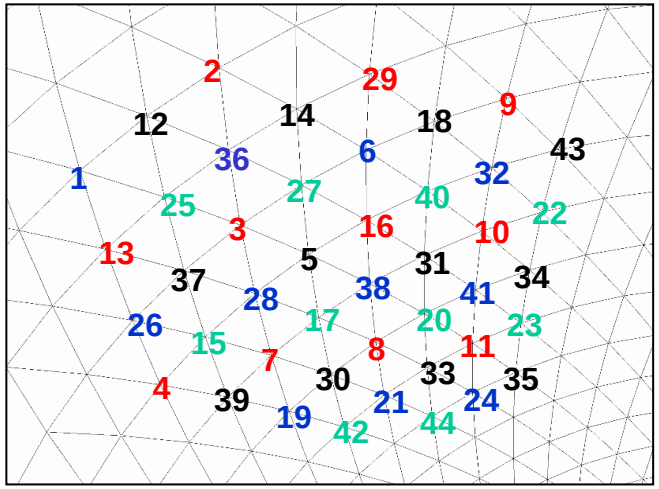
Multicolor Point-Implicit Solver: Basics

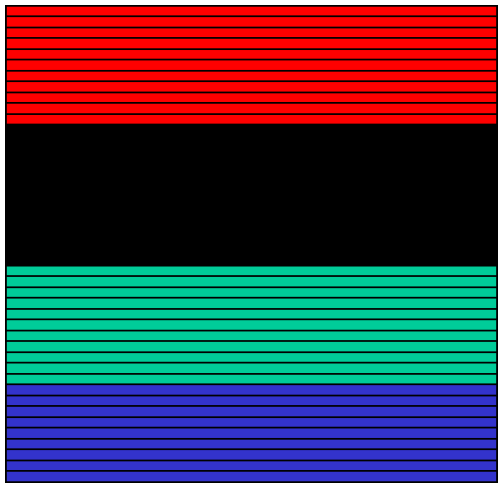
- Implicit scheme results in linear systems of equations:
  - $A\,\Delta Q = R$, A is a sparse $n \times n$ block matrix
  - Typically 14-19 blocks per row
  - block is of size $nb \times nb$ (typically, nb = 5)
- Matrix A is segregated into two separate matrices:
  - $A \equiv O + D$, where O and D represent the off-diagonal and diagonal blocks of A
  - D is always stored in double precision (FP64)
  - O is typically stored in single precision (FP32), option for half-precision (FP16)
- Prior to performing each linear solve, each diagonal block D is decomposed in-place into lower and upper triangular matrices

Multicolor Point-Implicit Solver

Uses a series of multicolor point-implicit sweeps to form an approximate solution to $A\,\Delta Q = R$

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



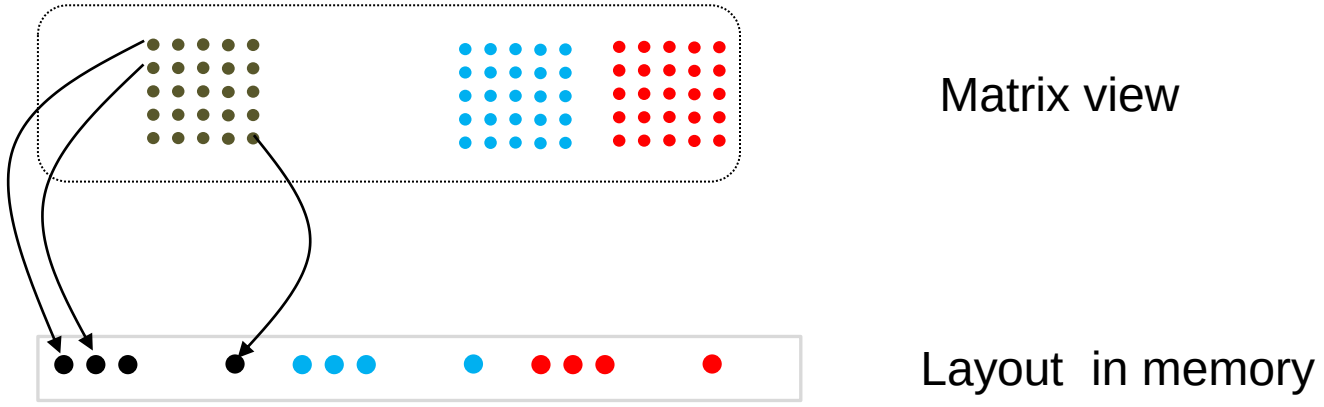


Algorithm 1 MULTICOLOR LINEAR SOLVER

$\Delta \mathbf{Q} = 0$
for $i \leftarrow 1$ **to** n_{iter} **do**
  for $c \leftarrow 1$ **to** n_c **do**
    $\Delta \mathbf{r} \leftarrow \mathbf{R_c} - \mathbf{O_c}\Delta \mathbf{Q}$
    $\Delta \mathbf{Q_c} \leftarrow \mathbf{D_c^{-1}}\Delta r$
  end for
end for

Matrix Storage and Performance Issues

- The dominant computation in the block-sparse linear solver is a block-sparse matrix-vector operation with a typical block size of 5 x 5
  - Memory bound computation so it is critical to utilize the memory bandwidth effectively
- The matrix is stored in a block CSR format, where the non-zero blocks in a row are stored contiguously in the memory
- A block is stored in a column-major order



Naive Implementation on NVIDIA V100

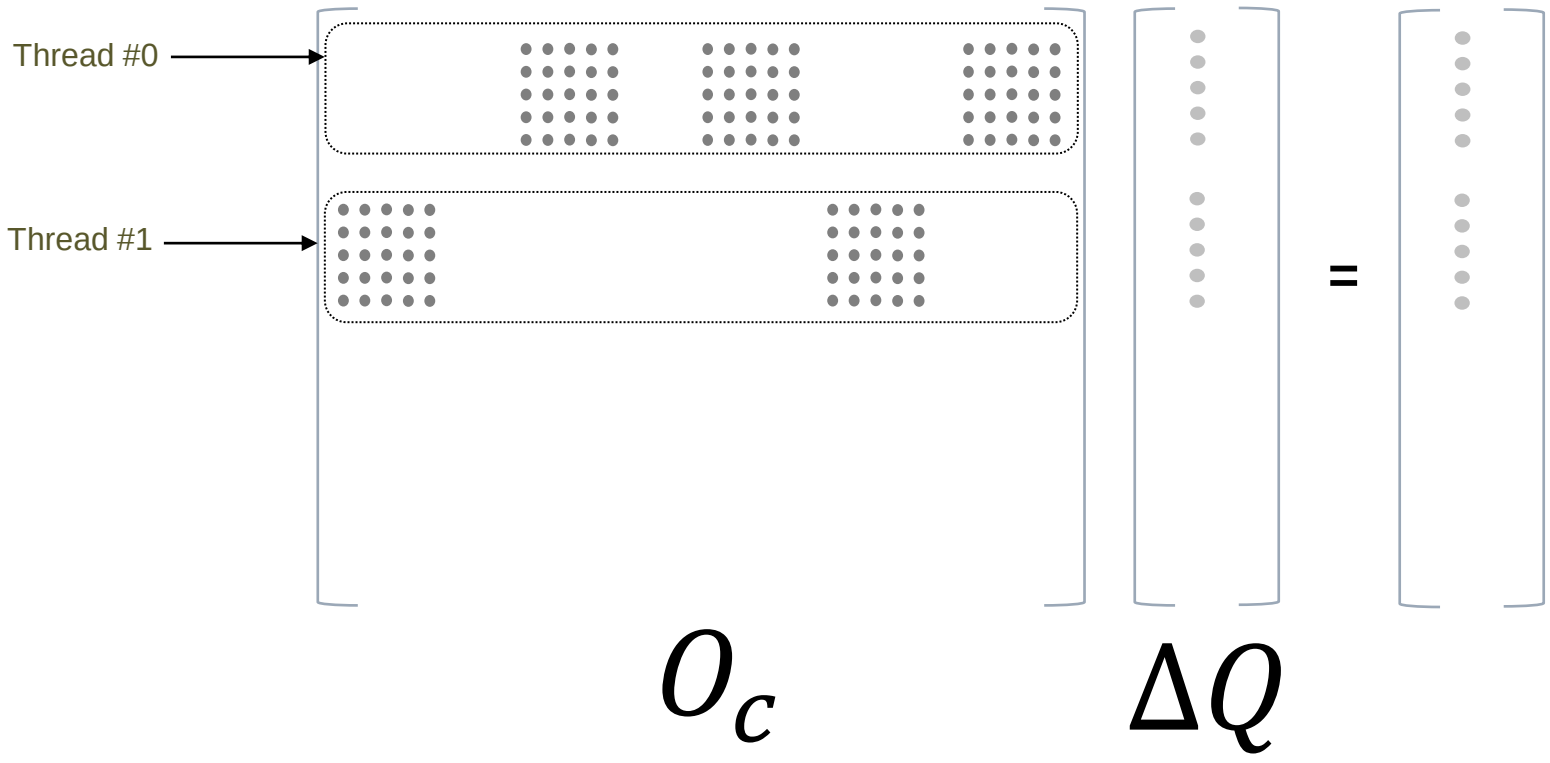
Memory access pattern for the naive implementation results in very poor utilization of memory bandwidth.

Consecutive threads are accessing memory locations that not consecutive.

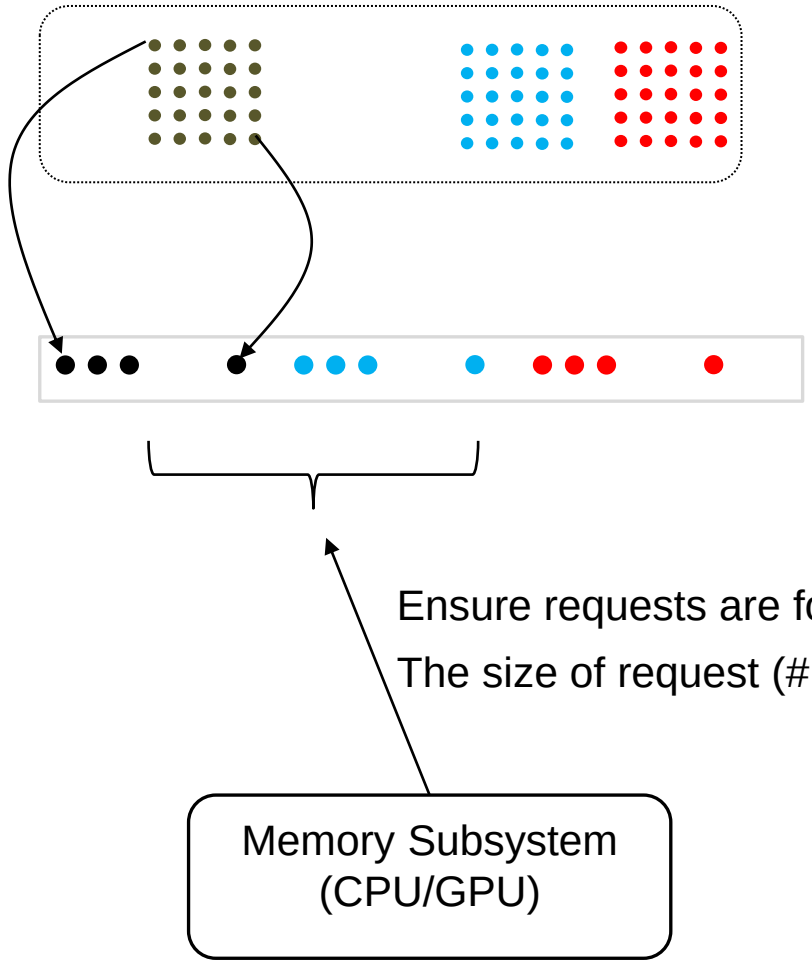
V100: ~500 ms ~ 70 GB/s

%Peak: 7.7% TP: 900 GB/s

Block Sparse Matrix Vector Operation



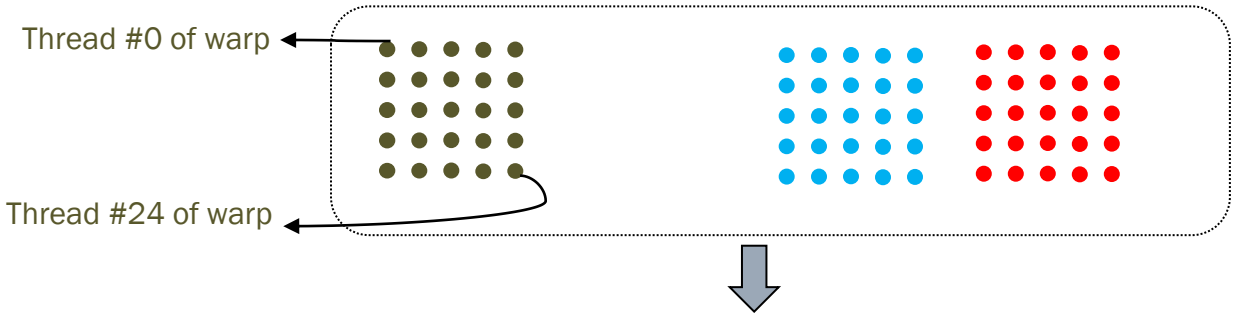
Optimization Issues for Multicolor Linear Solver



Optimization Issues on GPU

- GPU supports Single Instruction Multiple Thread (SIMT) model with a group of threads referred to as warp (or wavefront)
- The dimension of this thread group can vary from one GPU to another, and the group must process consecutive memory locations to achieve coalesced memory accesses
- This requires mapping the warp (or wavefront) to one or more blocks of a sparse matrix and restructuring the computation accordingly

Block Sparse Matrix Vector-FUN3D (CUDA)



Map a warp to a 5x5 block
Note: only 25 threads are active, 7 threads are inactive

```
k = threadIdx.x % 5;
l = threadIdx.x / 5;

for (j=istart-1; j < iend; j++) {
    colid = jam[j];
    fk += A_OFF(k,l,j)*DQ(l,colid-1);
}
//save partial aggregation in shared memory
sm_f[k][l][threadIdx.y] = fk;
.
.
```

Output is a 5x5 block of partial terms

alternate: avoid shared memory, use shuffle to aggregate

use one thread to aggregate 5 subcolumns in shared memory into a single column

```
// Reduction along the subcolumns
f1 = fk;
f1 = f1 + __shfl_sync(0xffffffff, fk, k + 1 * 5);
f1 = f1 + __shfl_sync(0xffffffff, fk, k + 2 * 5);
f1 = f1 + __shfl_sync(0xffffffff, fk, k + 3 * 5);
f1 = f1 + __shfl_sync(0xffffffff, fk, k + 4 * 5);
```

V100: ~48 ms ~ 716 GB/s
Performance improvement: 10x
%Peak: 79% TP: 900 GB/s

Challenges in Porting CUDA-Optimized Code on Pre-Production Intel GPU Hardware using Intel oneAPI

Terminology/Concepts Mapping

CUDA

CUDA programming model hides SIMD operations by exposing a physical thread as a number of logical threads.

Warp: is a physical thread consisting of 32 *consecutive* logical threads

Thread Block Size = number of physical threads * 32 (warp size)

CUDA kernel is written to operate on scalars. In other words, kernel can be viewed as a code that is executed by every logical thread.

Logical threads in a warp can take different paths in the program – SIMT model.

For effective utilization of device memory consecutive threads in a warp should access consecutive memory locations.

Terminology/Concepts Mapping

oneAPI

- Work Item is equivalent to CUDA threads

- Work Group is equivalent to CUDA thread block
  - Work Group Size = # of thread (physical) * SIMD sub-group size (< Max Work Group Size)
  - NOTE: Unlike CUDA block size where warp size for current models is 32, the SIMD sub-group size can be specified by the user as 8, 16, 32

- Nd range specifies work-items hierarchy
  - Global range (64, 8)
  - Local range for each work group (16, 8)

- NOTE: Intel oneAPI work group size is 16 x 8 and equivalent CUDA work group size is 32 x 4 . The second dimension of oneAPI work group range is the SIMD sub-group size, whereas in CUDA the first dimension of CUDA thread block is the SIMD sub-group size. This is important to know as coalesced memory load or vectorization is happening along different dimensions.

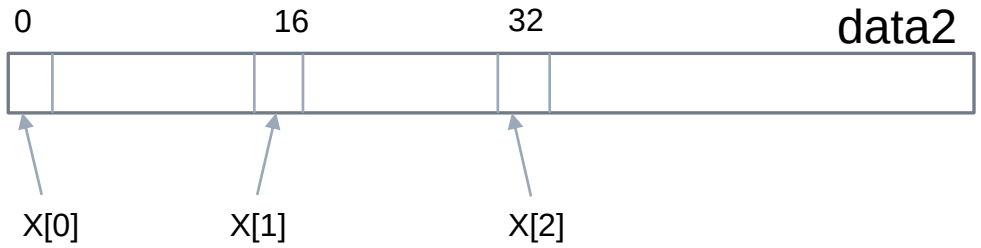
Terminology/Concepts Mapping

Intel oneAPI

- Subgroup Block Access: Enable a work-item to access a block of memory.
  - How should data be laid out to get the maximum benefit ?

Intel® graphics have instructions optimized for memory block loads/stores. So if work-items in a sub-group access a contiguous block of memory, we can use the sub-group block access functions to take advantage of these block load/store instructions.

intel::reqd_sub_group_size(16)
x = sg.load<8>(global_ptr(&(data2[base + 0])));




```
x = sg.load<8>(global_ptr(&(data2[base + 0])));
sg.store<8>(global_ptr(&(data[base + 0])), x);
x0 = sg.load<8>(global_ptr(&(data2[base + 128])));
sg.store<8>(global_ptr(&(data[base + 128])), x0);

out << " groupId = " << groupId
<< " i = " << i  << " base = " << base
<< " x0 = " << x[0]  << " x1 = " << x[1]
<< " x6 = " << x[6]  << " x7 = " << x[7]
<< sycl::endl;
```

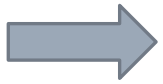
Where can block access be useful? Memory bound problems?

Optimized Intel oneAPI Implementation for Pre-Production Intel GPU Based on CUDA Optimized Code

CUDA Optimized Code

```
dim3 const nThreads(BLOCK_DIM_X, BLOCK_DIM_Y, 1);

__shared__ float sm_f[5][5][BLOCK_DIM_Y];
int const k = threadIdx.x % 5;
int const l = threadIdx.x / 5;
int n = start + blockIdx.x * blockDim.y + threadIdx.y - 1;
if ( (n < end) && (l < 5)) {
    int istart = iam[n];
    int iend = iam[n + 1];
    for ( j = istart; j < iend; j++) {
        jam0 = jam[j] - 1;
        fk += A_OFF(k, l, j) *  DQ(l, jam0);
    }
    sm_f[k][l][threadIdx.y] = fk;
}
__syncthreads();
```



Intel oneAPI code

```
cgh.parallel_for<class solver_point5>(
    sycl::nd_range<2> {sycl::range<2>(BLOCK_DIM_Y, gdimx),
         sycl::range<2>(BLOCK_DIM_Y, BLOCK_DIM_X)
}, kern );
```

[intel::reqd_sub_group_size(32)]

```
if ( (n < end) && (l < 5)) {
    fk = 0;
    jam1 = djam[istart-1] - 1;

    for (j = istart - 1; j < iend - 1; j++) {
      fk += A_OFF(k, l, j) *  DQ(l, jam1);
      jam1 = djam[j + 1] - 1;
    }
    fk += A_OFF(k, l, iend - 1) *  DQ(l, jam1);
    //fk += A_OFF(k, l, iend - 1) *  0.00001;
    sm_f[k][l][tidy] = fk;
}
item.barrier(sycl::access::fence_space::local_space);
```

Other Approaches Explored using Features Supported on the Intel Hardware

- Subgroup of various sizes from 8 to 32 with different mapping on row blocks of the sparse matrix
- Jason Sewall (Intel) also implemented a version that uses subgroup block access
- Explored SIMD SYCL Extensions

Resources

https://software.intel.com/content/www/us/en/develop/documentation/oneapi-gpu-optimization-guide/top.html

https://www.apress.com/gp/book/9781484255735

https://software.intel.com/content/www/us/en/develop/documentation/oneapi-dpcpp-cpp-compiler-dev-guide-and-reference/top/optimization-and-programming-guide/vectorization/explicit-vector-programming/explicit-simd-sycl-extension.html

Thanks to the Intel team who are very supportive in enabling and understanding optimization issues for programming Intel GPUs using oneAPI:

Kevin O'Leary, Zhiqi Tao, Jason Sewall, Scott Huck, Jeff Hammond, Alexander Tolikin, Mark Valcich, Jeff Rodgers, Tom Zahniser, and many others.

Conclusions

- Scientific kernels can greatly benefit from emerging high-performance architectures such as 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 compiler 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 to match the architecture, which is difficult to do in an automated way by the compiler 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.