Optimization of Ported CFD Kernels on Intel Data Center GPU Max 1550 Using oneAPI ESIMD

Mohammad Zubair
Intel oneAPI Center of Excellence
Old Dominion University

Christoph Bauinger and Xiao Zhu
Intel Corporation

Aaron Walden, Gabriel Nastac, and Eric Nielsen
NASA Langley Research Center

Background and Recent Applications

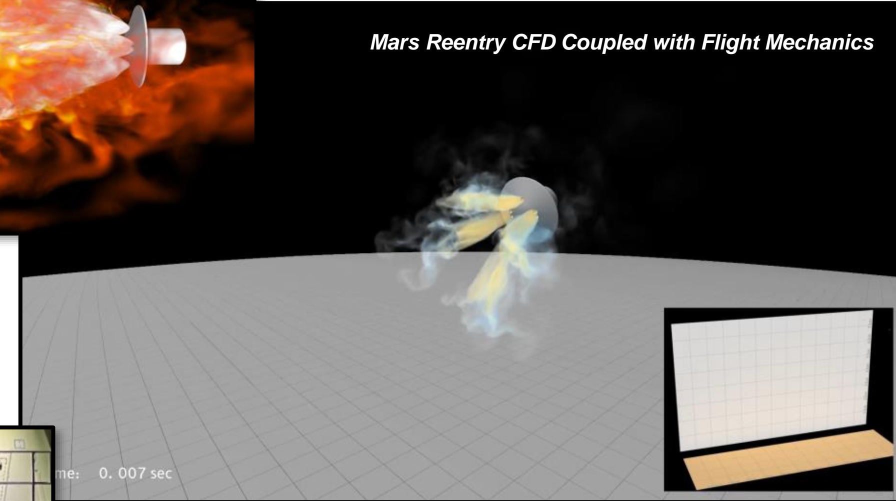
Background
- FUN3D solves the time-dependent compressible Navier-Stokes equations in fully implicit form on unstructured grids with thermochemical nonequilibrium and assorted turbulence treatments
- Lightweight abstraction over C++ for multi-architecture support using CUDA / HIP / SYCL (ESIMD)

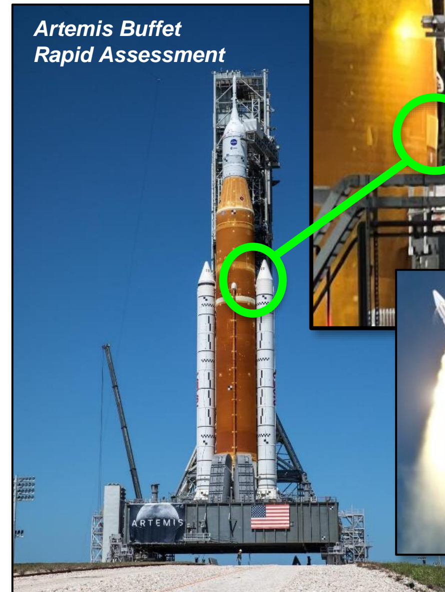
Retropropulsion for Human-Scale Mars Landers (Summit, Frontier)
- LOX / CH_4 rocket engines in Martian CO_2 atmosphere: 10-species, 19-reaction model (15 PDEs) on meshes of 7B elements; each sim produces ~1 PB of data
- Real-time coupling with ITAR-rated flight mechanics running remotely at NASA

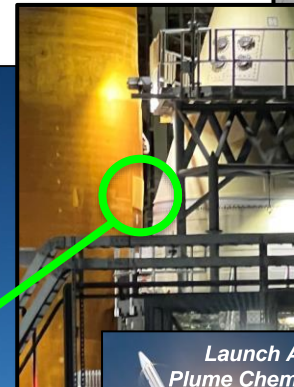
Wall-Modeled LES for High-Lift (Summit, Frontier, NASA)
- Essential for industry goal of Certification by Analysis

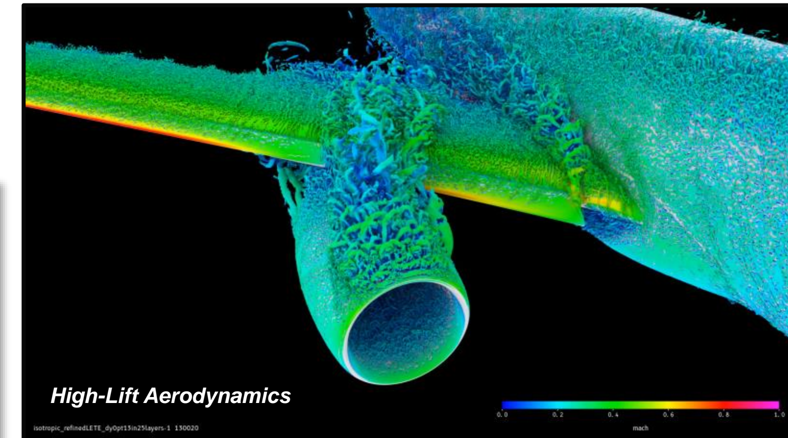
Artemis Program (NASA)
- Rapid 911 analysis supporting first launch
- Launch abort plume chemistry

Game-changing impacts for nationwide user base across capacity- and capability-class applications
- ***Faster turnaround / more sims / higher fidelity***
- ***Dramatic power reductions and space savings in the data center***









Outline

- Efforts over the past three years have focused on porting CUDA-optimized kernels to Intel oneAPI SYCL for use on Intel GPU hardware
- Suboptimal SYCL performance was observed on the new Intel data center GPU, with several challenges encountered:
  - High register spills
  - Memory latency
  - Poor vectorization
- Issues were addressed by implementing problematic kernels using Intel oneAPI's Explicit SIMD (ESIMD) API
- Performance comparisons shown for three kernels running SYCL and ESIMD on the Intel Data Center GPU versus CUDA-optimized versions running on NVIDIA V100 and A100 GPUs

Intel oneAPI / ESIMD

- We typically program a GPU using a programming model such as CUDA or SYCL, ignoring the underlying architecture’s “vector” aspects. Vectorization is generally left to the compiler.
- The ESIMD API is part of Intel oneAPI, and enables a programmer to write explicitly-vectorized kernel code
- ESIMD offers finer control over the vectorization compared to standard SYCL, which relies on the compiler for vectorization
- ESIMD also offers control over register usage, provides APIs for explicit memory load, store, and prefetch operations, and simplifies management of divergent branches in kernel code
- The main disadvantage of ESIMD versus SYCL is the lack of support for non-Intel hardware

Kernels Studied Here

Linear Solver

- A linear solver dominated by a block-sparse matrix-vector multiply operation, where performance is bound by the main memory bandwidth due to low arithmetic intensity

Matrix Assembly Based on a Hand-Differentiated (HD) van Leer Flux Jacobian

- A hand-differentiated version of a van Leer flux computation used to populate the block-sparse flux Jacobian matrix
- A significant amount of intermediate data results in high register pressure on GPU hardware

Matrix Assembly Based on Automatic-Differentiation (AD) for a Roe Flux Jacobian

- An automatic-differentiation version of a Roe flux computation used to populate the block-sparse flux Jacobian matrix

Notes

- *All three kernels have been previously optimized for NVIDIA GPUs and yield performance close to the theoretical peak*
- *All results shown on Intel GPUs have been performed using a single tile*
- *Here, the HD and AD linearization approaches are applied to different flux schemes and therefore cannot be compared directly. Such work is the focus of a paper to be presented next month at HiPC in Asia.*

Optimized CUDA Implementation

- Map a warp to multiply a row block of the block-sparse matrix by the appropriate vector elements

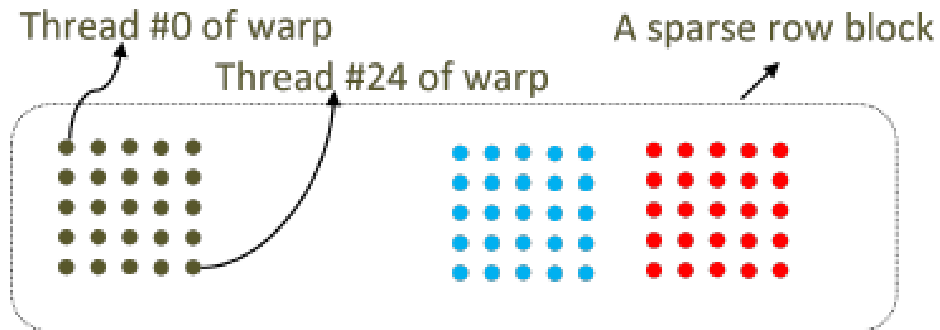
SYCL Implementation

- A straightforward conversion of the optimized CUDA implementation, where a sub-group of size 32 multiplies a row block of the block-sparse matrix by the appropriate vector elements

Observation

- Suboptimal performance of the SYCL implementation
- Motivates development of an ESIMD implementation to improve the performance

Linear Solver in CUDA and SYCL




```
1   int const k = threadIdx.x % 5;
2   int const l = threadIdx.x / 5;
3   int n = start+blockIdx.x*blockDim.y+threadIdx.y;
4   if ( (n < end) && (l < 5)) {
5       int istart = iam[n];
6       int iend   = iam[n+1];
7       fk = 0;
8       for ( j = istart; j < iend; j++) {
9           jam0 = jam[j];
10          fk += A_OFF(k,l,j) * DQ(l,jam0);
11      }
12      sm_f[k][l][threadIdx.y] = fk;
13  }
14  __syncthreads();
```

- Assignment of a warp to process a 5 x 5 block with coalesced memory accesses
- The 25 active threads process a block row one block at a time and aggregate partial results into a 5 x 5 block
- Columns of the aggregated block are reduced using shuffle instructions or shared memory (not shown)

- ***The CUDA implementation of this approach achieves a memory bandwidth of 710 GB/s on an NVIDIA V100, or 79% of the theoretical peak***
- ***The SYCL implementation of this approach achieves a memory bandwidth of 515 GB/s on an Intel GPU, or just 31% of the theoretical peak***

Linear Solver in ESIMD

```
1   col0 = icol32[0]-1;
2   // loop over non-zero blocks in a row
3   for (int j=istart; j<iend; j++) {
4     mv0 = block_load<float, 16>(a_off + j*25);
5     mv1 = block_load<float, 16>(a_off + j*25 + 15);
6     // load vector elements and replicate
7     dq8 = block_load<float, 8>(dq + col0*5);
8     dq25 = dq8.replicate_vs_w_hs<5, 1, 5, 0>(0);
9     col0 = icol32[j-istart+1]-1;
10    qv15 += mv0.select<15,1>(0) * dq25.select<15,1>(0);
11    qv10 += mv1.select<10,1>(0) * dq25.select<10,1>(15);
12  }
13
14  // aggregate 25 elements into a sub-vector of size 5
15  fr5s = qv15.select<5,1>(0) + qv15.select<5,1>(5) +
16    qv15.select<5,1>(10) + qv10.select<5,1>(0) + qv10.select<5,1>(5);
```

Load 16 elements into vector register mv0. Note that only 15 elements will be used.

Load 16 elements into vector register mv1. Note that only 10 elements will be used.

A partial result consisting of 25 elements is held by the two registers qv15 and qv10.

- ***Prefetching instructions to hide memory latencies are not shown here; see paper***
- ***The ESIMD implementation achieves a memory bandwidth of 1095 GB/s on an Intel GPU, or 67% of the theoretical peak***

Linear Solver Performance Summary

Architecture and Implementation	Time (ms)	Bandwidth (GB/s)	% of Peak Bandwidth
Intel GPU, SYCL	**66**	**515**	**31%**
Intel GPU, ESIMD	**31**	**1095**	**67%**
Intel GPU, Optimized SYCL	**34**	**1017**	**62%**
NVIDIA V100, CUDA	**48**	**710**	**79%**
NVIDIA A100, CUDA	**31**	**1100**	**71%**
NVIDIA A100, CUDA with L2 residency control	**27**	**1253**	**81%**

- The baseline SYCL implementation was improved through the use of prefetching intrinsics, manual loop strength reductions, and use of unreleased engineering versions of the Intel compiler and driver
- However, that implementation is significantly more intricate and thus harder to develop and maintain, and lacks portability due to reliance on Intel GPU-specific prefetching intrinsics

SYCL Implementation of HD Flux Jacobian

- Initial port of the CUDA-optimized kernel for the hand-differentiated van Leer flux performed poorly on the Intel GPU
  - Execution time on the Intel GPU was 162 ms compared to 3.6 ms on the NVIDIA A100 GPU
- The Intel oneAPI Toolkit release 2023.1 compiler generates register spills for the Intel GPU
  - We do not observe register spills for NVIDIA V100 / A100
- By leveraging the large General Register File (GRF) compiler option that trades off hardware threads for more registers per thread, performance improved by more than a factor of fourteen

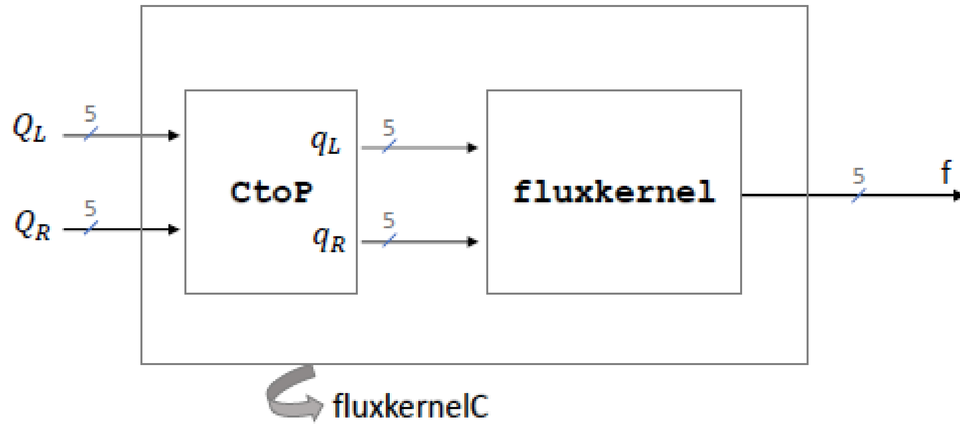
ESIMD Implementation of HD Flux Jacobian

- We vectorize across edges with a vector size of eight; i.e., a hardware thread processes eight edges concurrently
  - Care must be taken to avoid race conditions and divergence
  - Atomics used to avoid race conditions
  - Divergence handled efficiently by creating masks and through use of the ESIMD *merge* API
- To improve ESIMD performance, we also distribute the work inside the main loop between two hardware threads, thereby increasing parallelism
- We assume the eight edges being processed concurrently usually execute the same branch of physics based on a local Mach number condition

HD Flux Jacobian Performance Summary

Architecture and Implementation	Time (ms)
Intel GPU, SYCL	161.7
Intel GPU, SYCL with GRF option	11.4
Intel GPU, ESIMD	3.8
NVIDIA V100, CUDA	7.8
NVIDIA A100, CUDA	3.6

Automatic Differentiation of Flux Function



($\boldsymbol{Q}_L$,$\boldsymbol{Q}_R$) are conservative variable inputs that are converted to primitive variables ($\boldsymbol{q}_L$,$\boldsymbol{q}_R$) by the function CtoP. The flux kernel uses ($\boldsymbol{q}_L$,$\boldsymbol{q}_R$) to compute the flux.

Algorithm 4 FLUX-DERIVATIVEC(G, Aoff, Adiag)

1: Input: Grid G
2: **for** $i \leftarrow 1$ **to** n_{edges} in G **do**
3: **for** $j \leftarrow 1$ **to** 10 **do**
4: $Q_L, Q_R \leftarrow$ getConservative(G, i)
5: $dir \leftarrow$ getDirection(G, i)
6: $Q_{L_D}, Q_{R_D} \leftarrow$ initDual(Q_L, Q_R, j)
7: $localFlux_D \leftarrow$ fluxkernelC$_D$(Q_{L_D}, Q_{R_D}, dir)
8: updateAoff(Aoff, $localFlux_D, G, i$)
9: updateAdiag(Adiag, $localFlux_D, G, i$)
10: **end for**
11: **end for**
12: **return** Aoff, Adiag

Forward mode: Use of simple dual numbers with ten threads per edge

The use of multivariate dual numbers with two threads per edge was found to be most efficient. Using multivariate dual numbers of dimension 5, we can compute flux derivatives with respect to five Q_L (Q_R) variables with one call to the AD version of the flux routine.

SYCL Implementation of AD Flux Jacobian

- A straightforward translation of the optimized CUDA for the AD kernel yields SYCL code which performs poorly: 304 ms on Intel GPU vs 7.8 ms on NVIDIA V100 GPU
- The main culprit is register spills on the Intel GPU
  - By default, the Intel GPU provides 64 32-bit registers per work item for a sub-group of size 32, or one quarter of that available on the NVIDIA V100 GPU
- Large GRF mode, along with a reduced subgroup size of 16, eliminated the register spills and improved the performance to 14.5 ms.

AD Flux Jacobian Performance Summary

Architecture and Implementation	Time (ms)
Intel GPU, SYCL	14.5
Intel GPU, ESIMD	5.6
NVIDIA V100, CUDA	7.7
NVIDIA A100, CUDA	5.0

Summary

On any emerging architecture, achieving performance close to the theoretical hardware specifications requires careful analysis of the underlying architecture and design/implementation of algorithms to map to the hardware

For complex kernels, current high-level tools and programming models often cannot deliver this performance, so one may need to explore lower-level approaches

- Suboptimal performance on the Intel Data Center GPU Max 1550 was mainly due to register spills, memory latency, and poor vectorization; addressed by implementing the kernels using oneAPI ESIMD
- Performance close to the theoretical peak can be achieved using oneAPI ESIMD
- Single-tile performance of the Intel GPU using ESIMD is close to that of the NVIDIA A100 GPU
- Intel values this collaboration highly, as it is helping to improve SYCL in terms of both features and tools
    - Availability of prefetching, addressing register limitations, improving shuffle instructions, improved SYCL code generation (automatic loop strength reductions, minimizing instructions, automatic block loads, etc.)

Hardware and Acknowledgments

All Intel GPU performance* results presented in this paper were generated on an internal system with an Intel® Data Center GPU Max 1550 hosted at Intel. For compilation, we used an unreleased Intel® oneAPI DPC++/C++ Compiler 2024.0.0 (2024.0.0.20230713).

This effort has been sponsored by the Intel oneAPI Center of Excellence located at Old Dominion University and the NASA Transformational Tools and Technologies (TTT) Project of the Transformative Aeronautics Concepts Program under the Aeronautics Research Mission Directorate.

We are grateful for the advice and guidance from Pierre Boudier, Geoff Lowney, and Ben Ashbaugh at Intel.

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under contract DE-AC02-06CH11357. We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation at Argonne National Laboratory. This research also used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

*Legal Notices and Disclaimers. Performance varies by use, configuration and other factors. Learn more on the Performance Index site www.Intel.com/PerformanceIndex. Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. No product or component can be absolutely secure. Your costs and results may vary. Intel technologies may require enabled hardware, software or service activation. ©Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.