U.S. NAVAL RESEARCH LABORATORY

\Orchestrating a brighter world NEC

# NEC Vector Engine Performance with Legacy CFD Codes

Yu Yu Khine[1], Hiroshi Daikoku[2], Steve Frazier[3], Raghunandan Mathur[4], Takeshi Miyako[2], Keith Obenschain[1], Gopal Patnaik[5], Deepak Pathania[4], Kevin Peterson[3], Robert Rosenberg[1], Gursimranjit Singh[4]

[1] U.S. Naval Research Laboratory
[2] NEC Corporation
[3] Hewlett Packard Enterprise
[4] NEC Technologies India
[5] Syntek Technologies

## Introduction

- From the 1970's to early 1990's the U.S. DoD developed codes to take advantage of vector supercomputers (Cray 1 – Cray T90). These codes are still in use!
- The NEC Vector Engine (VE) is an up-to-date, high-performance version of the vector computer perfected by CRAY in the 1990s
- It can provide a 3-6X performance increase over conventional computers for legacy CFD simulation models
- The NEC VE provides an opportunity to make full use of these well-validated codes by giving them a much-needed performance boost *with some optimization*

## Objectives

- The goal is to demonstrate the capabilities of the NEC Vector Engine for a legacy CFD code, especially FDL3DI developed at U.S. Air Force Research Laboratory (AFRL)
- Supported by the DoD's Foreign Comparative Testing program whose goal is to transition innovative foreign technology into existing and future DoD programs, the objectives of the project are:
  - To benchmark standardized codes such as NAS parallel benchmarks, machine learning, etc. to study the performance of VE
  - To qualitatively assess the ease of use of the VE and to quantify the speedup over conventional Intel Xeon and recently available systems such as AMD EPYC

## Collaboration with NEC, HPE and AFRL

- Collaborate with NEC consultants on optimization of codes and mentoring on the use of the platform
  - Reach back to NEC hardware and software teams
- Work with Hewlett Packard Enterprise (HPE) on single and cluster designs using NEC Vector Engine cards
- Weekly meetings and sometimes daily discussions with NEC and HPE as needed
- Communicate with developer of FDL3DI code, Dr. D. Garmann at AFRL, on guidance in optimizing the code for NEC system

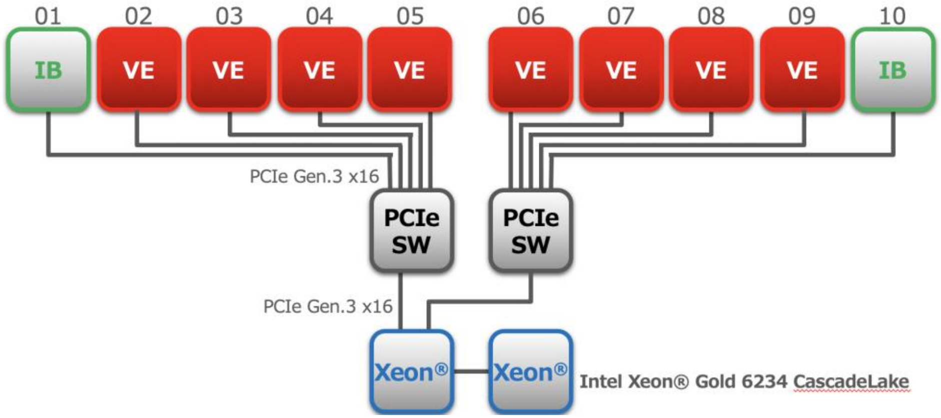
# NEC SX-Aurora TSUBASA Vector Engine (VE)

- NEC has been producing vector architectures since 1983
- Each VE has 8 cores, a total of 2.15 double precision TFLOPS
- Vector Register has 256 eight-byte elements
- The processor has the world's first implementation of six 3D-stacked High Bandwidth Memory modules with a total 48GB
- A Scalar Processing Unit handles non-vector instructions on each of the cores
- Runs C/C++/Fortran with MPI – **Codes primarily run on NEC VE hardware**

# VE Cluster Design

- Each Vector Host (VH) can host up to 8 VEs, clusters of VH can scale to arbitrary number of nodes to form a supercomputer.

HPE Apollo 6500 Gen10
8 VE Server



U.S. NAVAL RESEARCH LABORATORY

# Architectures Evaluated

\Orchestrating a brighter world NEC

| | Intel Xeon Platinum 8260 | AMD EPYC 7702 | NEC VE |
|---|---|---|---|
| Cores/Pipes per socket/device | 24 | 64 | 8 |
| Sockets/Devices per Node | 2 | 2 | 2 |
| Memory Technology | DDR4-2933 (six channels) | DDR4-3200 (eight channels) | HBM2 (six modules) |
| Theoretical Memory Bandwidth per device | 141 GB/sec | 204.8 GB/sec | 1.2 TB/sec |
| LLC Cache Size | 35.75MB | 256MB | 16MB |
| Process Technology | Intel/14nm | TSMC/7nm | TSMC/16nm |

U.S. NAVAL RESEARCH LABORATORY

## Completed Benchmarks

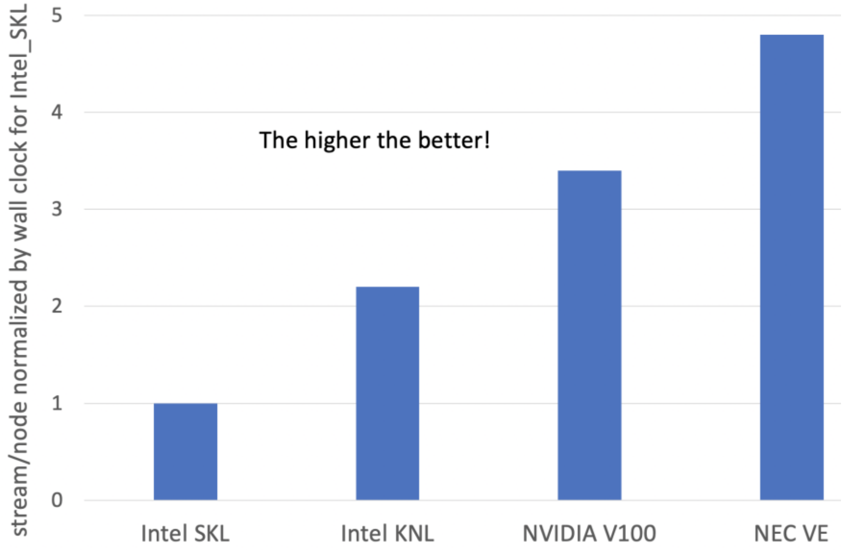
Orchestrating a brighter world NEC

- STREAM benchmark designed to measure sustainable memory bandwidth
- SGEMM and SAXPY routines in standardized NAS Parallel Benchmarks
- Standard molecular dynamics simulation
- NRL developed CFD code, FAST3D
- AFRL developed CFD code, FDL3DI
  - Timings are compared with those on Intel Xeon and AMD EPYC systems
  - Detailed profiling of FDL3DI on Intel, AMD and NEC systems

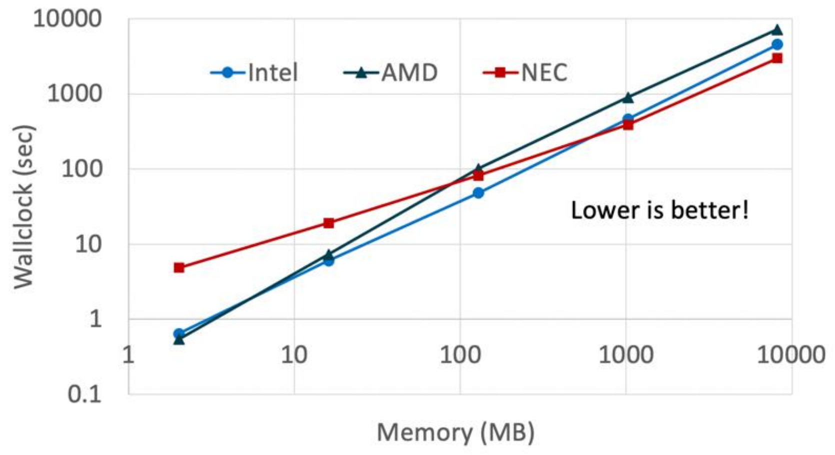
U.S. NAVAL RESEARCH LABORATORY

# Selected Benchmarking Results

\Orchestrating a brighter world NEC



- Performance of STREAM on various systems
- NEC VE outperforms Intel and NVIDIA



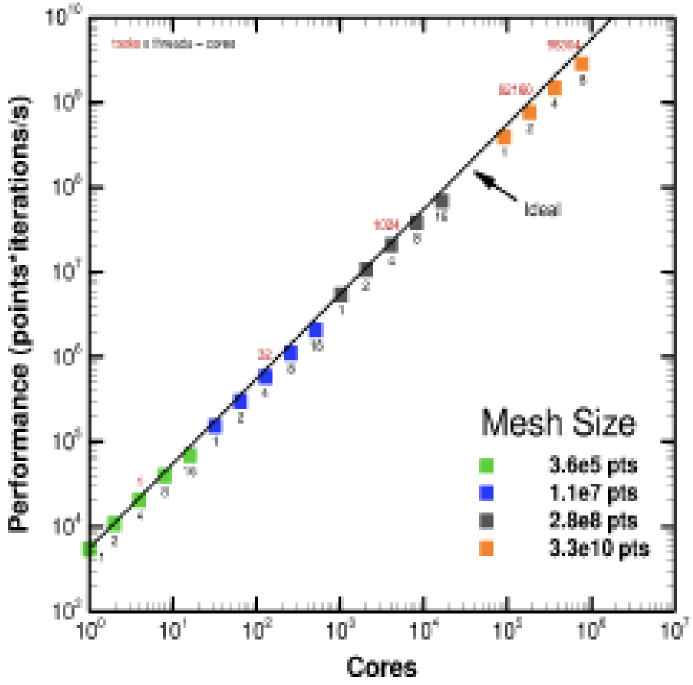
- FAST3D (CFD) code on Intel and NEC
- Code efficiently uses cache, main memory bandwidth bound on larger problems

# FDL3DI Overview

- Powerful, high-order, structured, overset CFD solver developed at AFRL
- Scalable and efficient
- Implicit large eddy simulation (LES) capability
- Compact scheme with filtering, hybrid shock capturing, high-order interpolation, hole handling
- Recent improvements in FDL3DI:
  - Fortran 90 with MPI-I/O
  - Hybrid MPI/OpenMP implementation
  - Robust hole-cutting and scheme adaption
  - Algorithmic enhancements via filter compact delta formulation

# Applications and Scaling

- Shock/boundary layer interaction in front of canonical shapes
- Wing-vortex aerodynamics
- Flow control for laminar flow airfoils
- Linear scaling for large mesh sizes



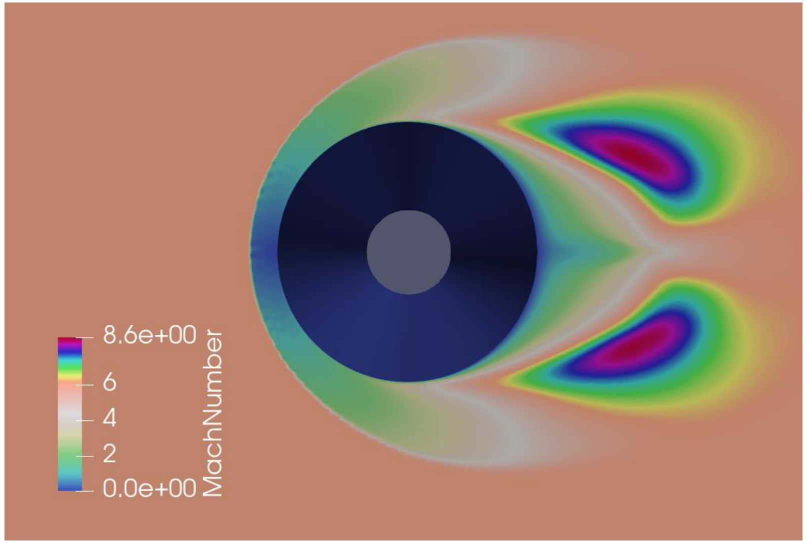
*Ref*: *"High-Order Overset CFD Simulations Using FDL3DI", S. E. Sherer, 13th Overset Grid Symposium, Mulkiteo, WA, 17-20 October 2016.*

U.S. NAVAL RESEARCH LABORATORY

# Test Case for FDL3DI

\Orchestrating a brighter world NEC

- A classic cylinder in a free stream example
- Freestream Mach number = 6
- Zero angle of attack -- flow from left
- Shock Sensor – DUCROS
- Three problem sizes: $128^3$, $256^3$, $480^3$
- Time step size = 0.001
- Timings taken for 101-150 time steps



Mach number representation of flow past cylinder

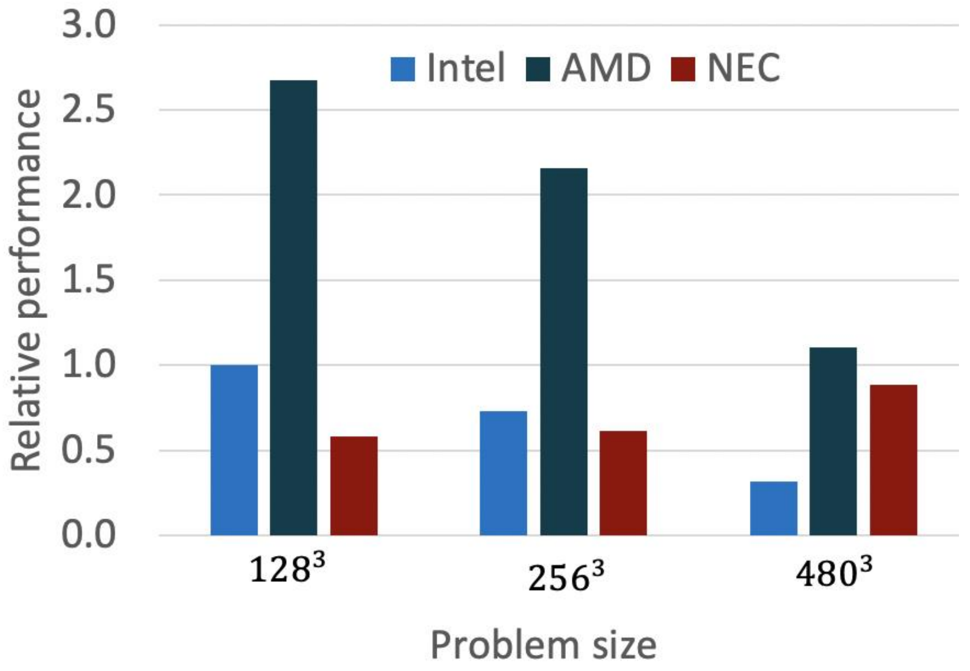
## Problem Sizes for FDL3DI Code

- FDL3DI run on NEC VE node (2 VE Accelerators) with eight MPI Ranks using hybrid MPI/OpenMP
- For eight MPI ranks, each dimension is divided by two so the maximum vector length is half the problem dimension plus guard cells
- $480^3$ case was selected to increase the average utilization of vector pipes
  - Vector length for this problem is close but does not exceed 256
  - Exceeding a vector length of 256 (e.g. problem size $512^3$) results in poor performance due to inefficient use of VE

| Problem Size | Block Size | Max Vector Length |
|---|---|---|
| $128^3$ | $64^3$ | 68 |
| $256^3$ | $128^3$ | 132 |
| $480^3$ | $240^3$ | 244 |

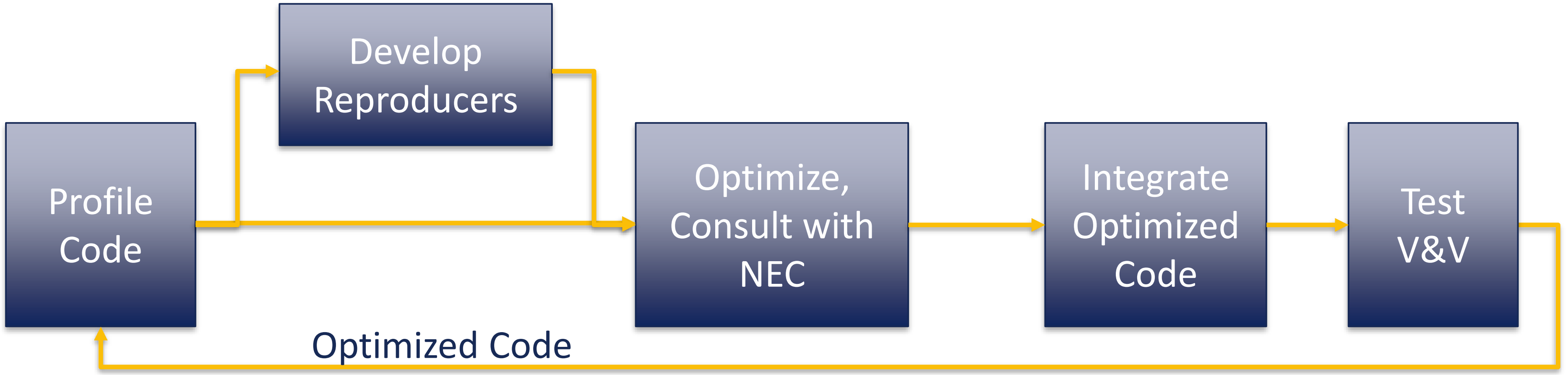
# Baseline Timings of FDL3DI

- Run without code modifications
- AMD EPYC's large cache (Last Level Cache: 256MB) makes it a strong competitor for smaller problem sizes
- Main memory bandwidth becomes important at larger problem sizes



Performance normalized by $128^3$ case run on Intel Xeon

## FDL3DI Optimization Process



- The most time-consuming routines were optimized for better performance on VE
  - Started with the tridiagonal solver and other time-consuming routines
- Optimized routines integrated back into codebase resulting in optimized FDL3DI

U.S.NAVAL RESEARCH LABORATORY

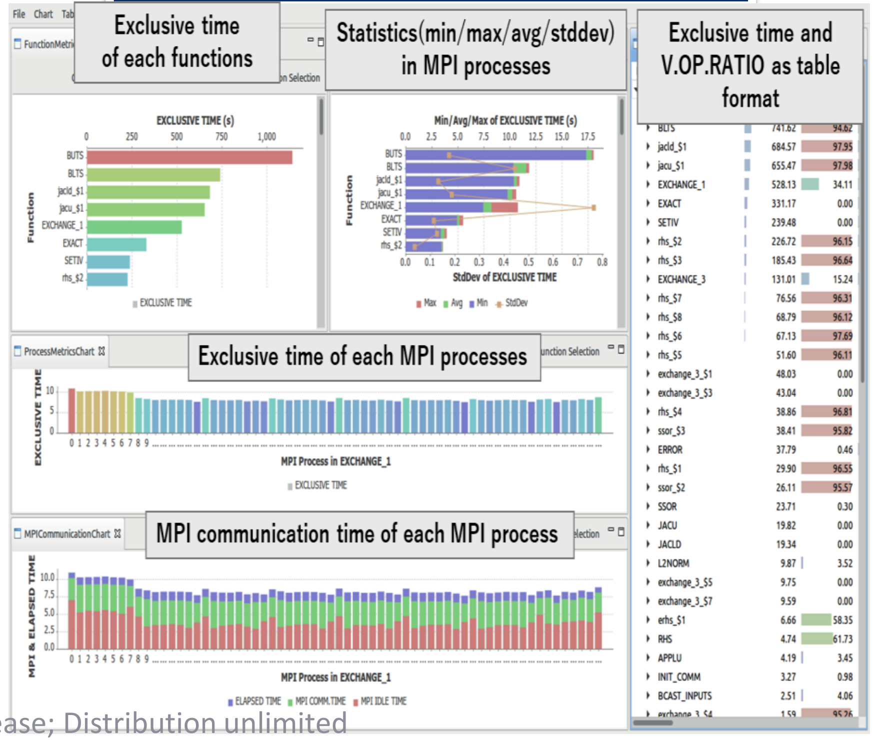
# NEC Development Environment

\Orchestrating a brighter world NEC

## Programming Framework

| | |
|---|---|
| C/C++ | • ISO/IEC 9899:2011 (aka C11)<br>• ISO/IEC 14882:2014 (aka C++14) |
| Fortran | • ISO/IEC 1539-1:2004 (aka Fortran 2003)<br>• ISO/IEC 1539-1:2010 (aka Fortran 2008) |
| OpenMP | • Version 4.5 |
| Library Support | • MPI Version 3.1 (fully tuned for Aurora architecture)<br>• Numeric libraries (BLAS, FFT, Lapack, etc) |
| ML Libraries | • TensorFlow<br>• Frovedis (spark, scikit-learn) |
| Tools | • GNU Debugger (gdb)<br>• Eclipse Parallel Tools Platform (PTP)<br>• FtraceViewer / PROGINF |

## In-house Profilers



| | | |
|---|---|---|
| BLTS | 741.62 | 94.62 |
| jacld_$1 | 684.57 | 97.95 |
| jacu_$1 | 655.47 | 97.98 |
| EXCHANGE_1 | 528.13 | 34.11 |
| EXACT | 331.17 | 0.00 |
| SETIV | 239.48 | 0.00 |
| rhs_$2 | 226.72 | 96.15 |
| rhs_$3 | 185.43 | 96.64 |
| EXCHANGE_3 | 131.01 | 15.24 |
| rhs_$7 | 76.56 | 96.31 |
| rhs_$8 | 68.79 | 96.12 |
| rhs_$6 | 67.13 | 97.69 |
| rhs_$5 | 51.60 | 96.11 |
| exchange_3_$1 | 48.03 | 0.00 |
| exchange_3_$3 | 43.04 | 0.00 |
| rhs_$4 | 38.86 | 96.81 |
| ssor_$3 | 38.41 | 95.82 |
| ERROR | 37.79 | 0.46 |
| rhs_$1 | 29.90 | 96.55 |
| ssor_$2 | 26.11 | 95.57 |
| SSOR | 23.71 | 0.30 |
| JACU | 19.82 | 0.00 |
| JACLD | 19.34 | 0.00 |
| L2NORM | 9.87 | 3.52 |
| exchange_3_$5 | 9.75 | 0.00 |
| exchange_3_$7 | 9.59 | 0.00 |
| erhs_$1 | 6.66 | 58.35 |
| RHS | 4.74 | 61.73 |
| APPLU | 4.19 | 3.45 |
| INIT_COMM | 3.27 | 0.98 |
| BCAST_INPUTS | 2.51 | 4.06 |
| exchange_3_$4 | 1.59 | 95.26 |

# NEC Development Environment Experience

- Installation via RPM’s straightforward
- Initial codes compiled without modification
- Encountered several compiler issues while optimizing codes on VE

  - Some cases of compiler aborts and cases of deviation from Fortran standards were reported and were promptly resolved in the next release of the compiler

  - Some incompatibilities of code with automatic vectorization are being discussed, some cases involved work-arounds

# NEC Development Environment Profilers

- NEC's profiler, FTRACE, is used to obtain performance information such as the processor usage and vectorization aspect of each function in a program, as well as user defined regions
  - Does not give loop-by-loop analysis and does not profile in-lined code
  - User defined regions provided a method for finer grain analysis
  - Adds overhead

- Installed and applied the profiler, Tuning and Analysis Utilities (TAU), on NEC system
  - Engaged with developer of TAU, Paratools, Inc. to support NEC VE
  - Instrumentation was too slow on NEC system (work in progress)

# Sample FTRACE Output

```
FREQUENCY EXCLUSIVE       AVER.TIME     MOPS    MFLOPS  V.OP  AVER.  VECTOR L1CACHE   CPU PORT VLD-LLC PROC.NAME
          TIME[sec]( % )    [msec]                      RATIO V.LEN    TIME    MISS   CONF HIT E.%

     1012   49.093( 24.0)   48.511  23317.2  14001.4  96.97  83.2  42.132   5.511    0.000   80.32   funcA
   160640   37.475( 18.3)    0.233  17874.6   9985.9  95.22  52.2  34.223   1.973    2.166   96.84   funcB
   160640   30.515( 14.9)    0.190  22141.8  12263.7  95.50  52.8  29.272   0.191    2.544   93.23   funcC
   160640   23.434( 11.5)    0.146  44919.9  22923.2  97.75  98.5  21.869   0.741    4.590   97.82   funcD
   160640   22.462( 11.0)    0.140  42924.5  21989.6  97.73  99.4  20.951   1.212    4.590   96.91   funcE
 53562928   15.371(  7.5)    0.000   1819.0    742.2   0.00   0.0   0.000   1.253    0.000    0.00   funcG

-----------------------------------------------------------------------------------------------------------
 54851346  204.569(100.0)    0.004  22508.5  12210.7  95.64  76.5 154.524  17.740   13.916   90.29  total

    62248   37.709( 18.4)    0.606   2200.2   1026.4   0.00   0.0   0.000   0.532    0.000   20.00   loop#1
     2032    4.834(  2.4)    2.379    415.8      0.0  28.61   6.3   4.098   0.246    0.000    0.00   loop#2
...
```

Goal: Identify bottleneck routines and optimize them to efficiently using Vector Engine Units

- Improve Vector Operation Ratio (V.OP) - code needs to vectorize!
- Improve Average Vector Length (Aver. V. Len) – longer vector length, better performance

U.S. NAVAL RESEARCH LABORATORY

## Optimization Techniques

\Orchestrating a brighter world NEC

### Vectorize

- Refactor serial routines to vector friendly implementations
- Removing vector inhibitions for a high vector operation ratio

### VREG

- A vector register (**vreg**) is a compiler directive supported by the NEC compilers that helps allocate a local array onto any available vector register
- During the code generation, the compiler assigns a dedicated vector register to the array
- Was used to increase data reuse – less pressure on memory subsystem

### Loop Collapse

- Address two or more dimensions as one long dimension (e.g.  A(m,n) becomes A(m*n,1))
- Increases effective vector length – better use of VE hardware

U.S. NAVAL RESEARCH LABORATORY

\Orchestrating a brighter world NEC

# Example of vreg

**ORIGINAL**

```
+------> DO m = 1, num_vars
|U----->  DO i = 3, ni-2
||V---->   DO k = ks, ke
|||          wrk(k,i,m,2) = 0.25 * wrk(k,i+2,m,1)+
|||                       & 0.50 * wrk(k,i+1,m,1)
||V----      END DO
|U-----     END DO
+------   END DO
```

- Vector registers are used for the computation with vector loads and stores in each iteration.
- Compiler automatically vectorizes inner loop.
- Compiler automatically unrolls the outer loop.

- Compiler directive that assigns specific arrays to vector registers.
- Block the outermost loop at 256 elements to ensure absolute usage of the vector registers.
- Arrays assigned to dedicated vector registers.
- The computation utilizes dedicated vector registers avoiding load-store latency.

**OPTIMIZED**

```
                !NEC$ vreg(wrk0)
                !NEC$ vreg(wrk1)

+------>DO ksv = ks, ke, 256
|        kev = MIN(ke, 256+ksv-1)
|
|+----->  DO m = 1, num_vars
||V---->   DO k = ksv, kev
|||          k_blk = k-ksv+1
|||     V    wrk0(k_blk) = wrk(k,1,m,1)
|||     V    wrk1(k_blk) = wrk(k,2,m,1)
||V----    END DO
||+---->   DO i = 3, ni-2
|||V--->    DO k = ksv, kev
||||          k_blk = k-ksv+1
||||
||||    V     wrk(k,i,m,2) = 0.25 * wrk0(k_blk)+
||||                       & 0.50 * wrk1(k_blk)
||||
||||    V     wrk0(k_blk) = wrk1(k_blk)
||||    V     wrk1(k_blk) = wrk2(k_blk)
|||V---     END DO
||+----       END DO
|+-----  END DO
```

U.S. NAVAL RESEARCH LABORATORY

# Example of Loop Collapsing

\Orchestrating a brighter world NEC

**ORIGINAL**

```
 REAL(KIND=8),DIMENSION(:,:,:,:), ALLOCATABLE :: wrk

V===> ALLOCATE(wrk(dim1,dim2,dim3,dim4),SOURCE=zero)

+---->  DO j = 1, dim1
|V--->   DO i = 1, dim2
||         wrk(i,j,1,4)= nx(i,j) + ny(i,j) + nz(i,j)
||         wrk(i,j,2,4)= nx(i,j) + ny(i,j) + nz(i,j)
||         wrk(i,j,3,4)= nx(i,j) + ny(i,j) + nz(i,j)
||         wrk(i,j,4,4)= nx(i,j) + ny(i,j) + nz(i,j)
|V---    END DO
+----   END DO
```

First two dimensions of the array are traversed using nested loops.

Declaration of multidimensional allocatable work array.

**OPTIMIZED**

```
 REAL(KIND=8),DIMENSION(:,:,:,:),pointer,
                              & contiguous :: wrk

 REAL(KIND=8),DIMENSION(:,:,:), pointer :: wrk1d

V===> ALLOCATE(wrk(maxd,maxd,nvar,6),SOURCE=zero)

      wrk1d(nsize,dim3,dim4) => wrk

      nsize = dim1 * dim2
V---> DO i = 1, nsize
|       wrk1d(i,1,4) = nx1d(i) + ny1d(i) + nz1d(i)
|       wrk1d(i,2,4) = nx1d(i) + ny1d(i) + nz1d(i)
|       wrk1d(i,3,4) = nx1d(i) + ny1d(i) + nz1d(i)
|       wrk1d(i,4,4) = nx1d(i) + ny1d(i) + nz1d(i)
V---  END DO
```

Nested loop collapsed into a single loop.

Pointing to original multidimensional array.

Declaration of a new pointer, targeted at treating two dimensions of `wrk` array as one long vector.

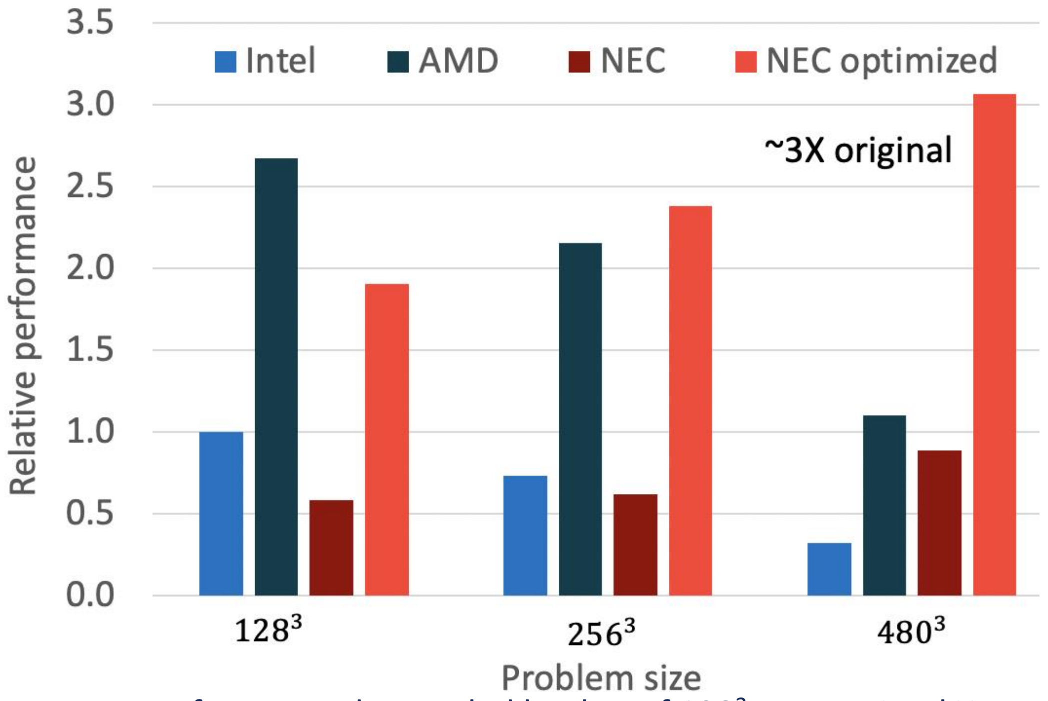
Declaration of a contiguous pointer to a multidimensional array.

U.S. NAVAL RESEARCH LABORATORY

# Single node comparison for Original and Tuned FDL3DI

\Orchestrating a brighter world NEC

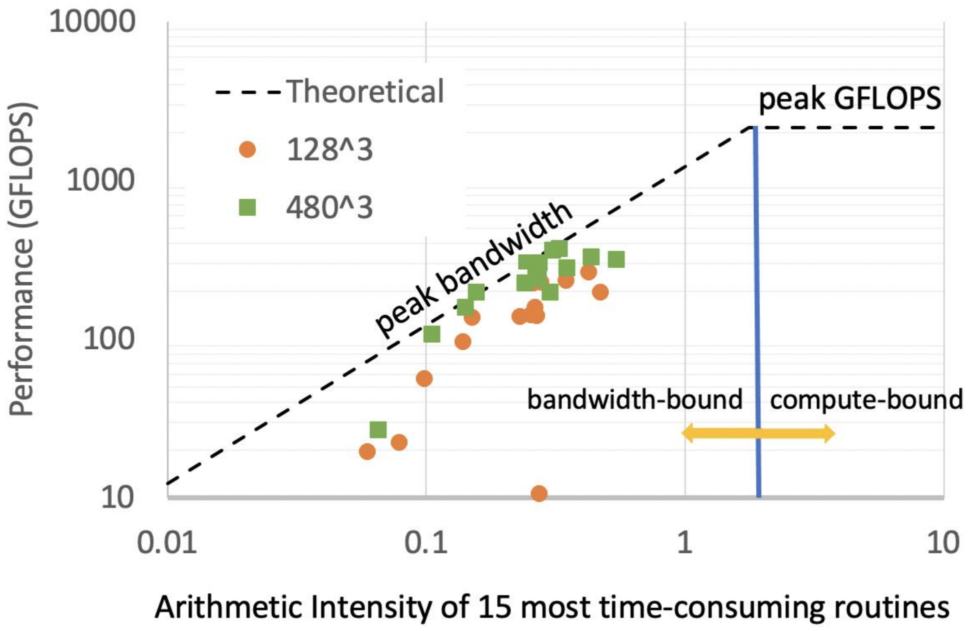
- Refactoring efforts significantly improved utilization of VE hardware resulting in significant performance gains for all problem sizes
- Use of VREG increases effective memory bandwidth



Performance data scaled by that of $128^3$ case on Intel Xeon

# Roofline Analysis of FDL3DI

- A Roofline Model is an easy way to visualize performance in relation to arithmetic intensity of the compute kernel
- CFD Codes are typically memory bound, top FDL3DI routines demonstrate this behavior
- $480^3$ case is near the Peak-bandwidth line
  - Tuned FDL3DI is making efficient use of VE
  - low arithmetic intensity limits performance to available memory bandwidth



# Verification

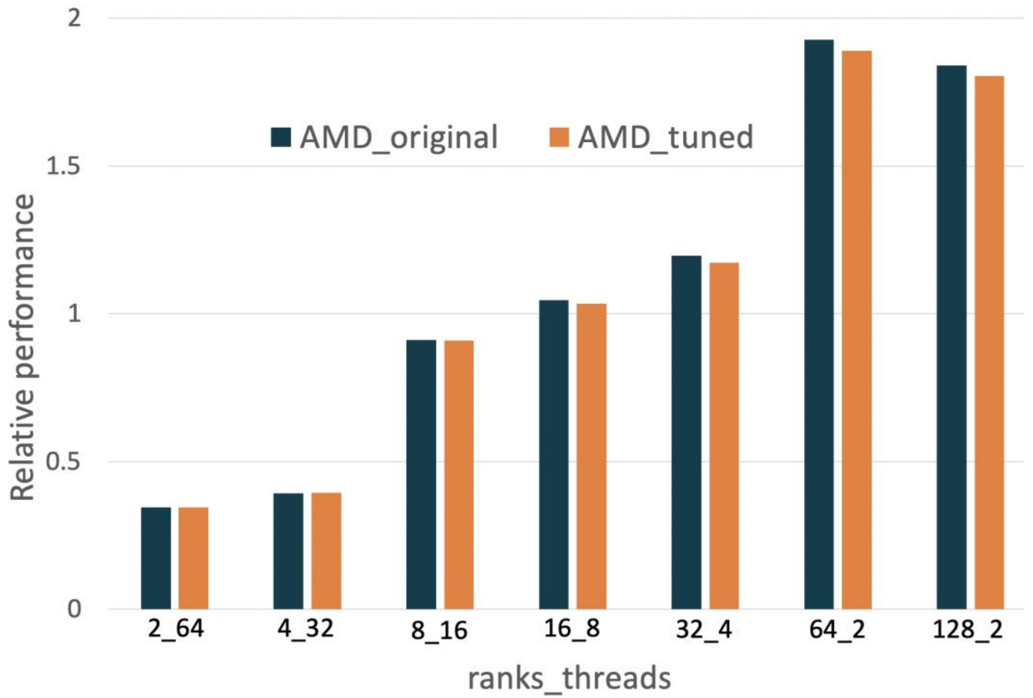
- Ran simulations on Intel Xeon, AMD EPYC and NEC Vector Engine systems
- Numerical results are visualized using Paraview
- Flow variables and forces are compared for accuracy for each simulation
- Results suggest $128^3$ is very coarse grid for this test case
- Results for $256^3$ and $480^3$ are very close for all the domain decompositions considered

U.S. NAVAL RESEARCH LABORATORY

## Baseline and tuned versions of FDL3DI on an AMD node for $256^3$ case

\Orchestrating a brighter world NEC

- Source code improvements tested on one node of AMD EPYC and Intel Xeon Systems
- No significant impact on the tuned FDL3DI code on AMD EPYC and Intel Xeon



## Conclusions

- Codes can run unchanged on the NEC VE, but optimization is required to take full advantage of the VE architecture
  - Codes that originally targeted vector architectures have been optimized for different architectures over the years: scalar code from later implementations needs to be optimized for vector architectures
  - Vector Engine requires efficient use of the 256 element double precision registers
  - Optimized codes more complex, but still quite readable by developers

U.S. NAVAL RESEARCH LABORATORY

## Conclusions Continued

\Orchestrating a brighter world NEC

- Codes limited by main memory bandwidth are good candidates for the current NEC VE architecture
- Other architectures are competitive depending on the problem characteristics:
  - Cache-friendly:  AMD EPYC
  - Short Vectors: SVE, AVX2, AVX512 Instruction Set Architectures
  - Optimization can mitigate (loop collapse, VREG)

## Future Work

- MPI Scaling beyond 2 VE’s
  - Eight NEC VE’s on upcoming development node
  - Larger problems put additional stress on MPI and communication
- Benchmark and profile development versions of FDL3DI
  - Changes to algorithms/implementation could change performance characteristics
  - Different numbers of OpenMP threads
- Examine other codes/relevant mini-apps (e.g. public-domain DOE mini-apps)
- Examine other profilers such as TAU
- Estimation of future NEC Vector Engine performance

# Acknowledgements

- This project is co-sponsored by the U.S. Department of Defense Foreign Comparative Testing Program within the Office of the Undersecretary of Defense for Research & Engineering
- Co-sponsored by U.S. DoD High Performance Computing Modernization Program
- Co-sponsored by the Office of Naval Research through the Naval Research 6.1 Materials Science Task Area
- Close collaboration with NEC consultants is also greatly appreciated
- Thanks to Dr. D. Garmann at the U.S. Air Force Research Laboratory on the guidance on the FDL3DI code