An Optimized Multicolor Point-Implicit Solver for Unstructured Grids on the ThunderX2 Arm Processor

Eric Nielsen and Aaron Walden
NASA Langley Research Center (LaRC)

Mohammad Zubair and Jason Orender
Old Dominion University (ODU)

John Linford
arm

https://fun3d.larc.nasa.gov

This research was supported in part by the NASA Langley Research Center High Performance Computing Incubator (LaRC HPCI) and the DoD HPCMP PETTT program.

Overview

- FUN3D
  - Widely-used CFD software suite (Fortran 90 with CUDA port)
  - Solves Navier-Stokes equations on mixed-element unstructured grids
  - Main computational kernel is a linear solver
  - Linear solver is essentially a sparse matvec that is memory bandwidth bound

- Our group pursues performance on emerging HPC architectures
  - Acquired a dual-socket 28-core ThunderX2 (TX2) node (56 total cores)
  - Benchmarked TX2 node using **solver mini-app** (Fortran) and STREAM
  - Developed a NEON vector intrinsic version of mini-app
  - NEON performance compared to AVX-512 intrinsic solver mini-app

FUN3D Overview

- FUN3D solves the Navier-Stokes equations of fluid dynamics using implicit time integration on general unstructured grids
- 2nd order finite volume discretization
- 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 Δq in linear system $A\Delta q = b$
6: $q \leftarrow q + \Delta q$
7: **end for**

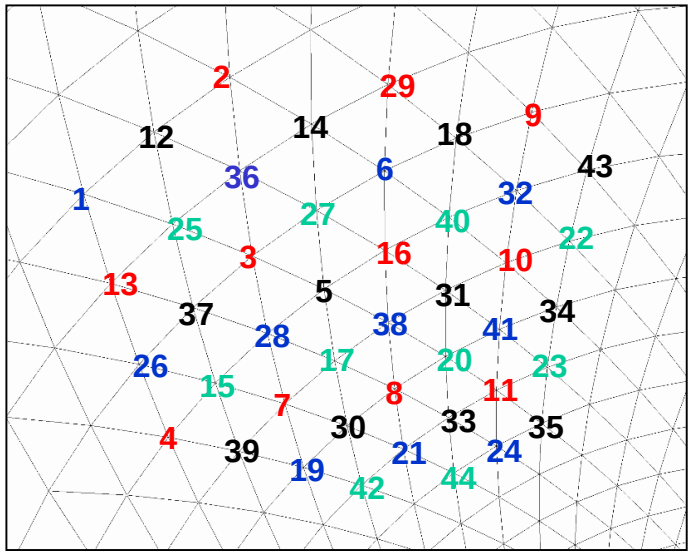
Multicolor Linear Solver: Basics

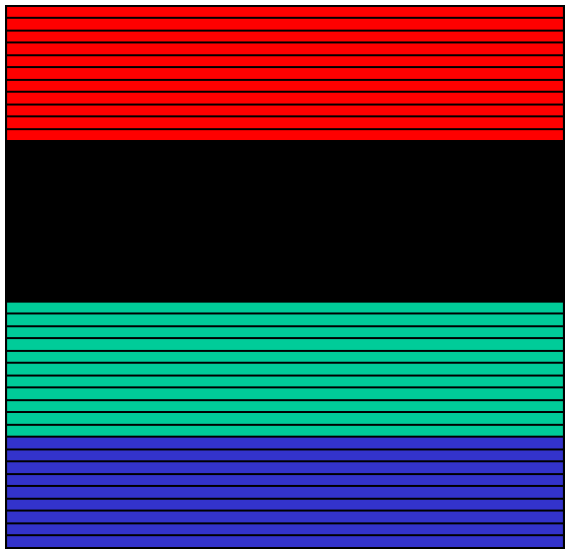
- Implicit scheme results in linear systems of equations:
  - $A\,\Delta q = b$, A is a sparse $n \times n$ block matrix
  - Typically 14-19 blocks per row
  - block is of size $nb \times nb$ (typically, nb = 5)
- Matrix A is segregated into two separate matrices:
  - $A \equiv O + D$, where O and D 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)
- Prior to performing each linear solve, each diagonal block D is decomposed in-place into lower and upper triangular matrices

Multicolor Linear Solver

FUN3D uses a series of multicolor point-implicit sweeps to form an apx. solution to **Ax = b**

- 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



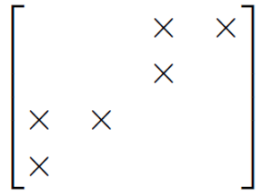


Algorithm 2 LINEAR SOLVER

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

Multicolor Linear Solver: Memory Layout

Sparse Structure of Matrix *O*



[x indicates a non-zero block]

CSR Storage for the Matrix to the left

$$
\begin{aligned}
ia &= [1, 3, 4, 6, 7] \\
ja &= [3, 4, 3, 1, 2, 1] \\
data &= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, \\
&\quad 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
\end{aligned}
$$

Matrix *O* with a 2×2 Block Size

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 3 & 5 & 7 \\
0 & 0 & 0 & 0 & 2 & 4 & 6 & 8 \\
0 & 0 & 0 & 0 & 9 & 11 & 0 & 0 \\
0 & 0 & 0 & 0 & 10 & 12 & 0 & 0 \\
13 & 15 & 17 & 19 & 0 & 0 & 0 & 0 \\
14 & 16 & 18 & 20 & 0 & 0 & 0 & 0 \\
21 & 23 & 0 & 0 & 0 & 0 & 0 & 0 \\
22 & 24 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

Multicolor Linear Solver: Challenges

- Indirect memory addressing (for vector Δq)
- Low arithmetic intensity (≈0.5 flops/byte) – memory bound on CPU and GPU
- The number of rows associated with a color, and thus the coarse-grained parallelism available, can vary significantly
- To support strong scalability, the single node performance for light workloads should be good

ThunderX2 Performance Overview

Slide (mostly) courtesy of arm TX2 presentation

- Fully pipelined execution units
  - ~560 GFLOP theoretical peak
  - 2xNEON 128-bit vector engines per core
  - 2x128-bit load/store units: either load 2x128-bit or load 1x128 and store 1x128
  - L2 can load or store two cache lines. L3 is exclusive of L2 (Victim cache)

- DRAM Memory Bandwidth
  - 8 memory channels
  - Up to 130GB/s per socket for STREAM Triad

- Adaptive stride and pattern matching algorithms ➔ as effective as non-temporal load/store

- Up to four-way SMT
  - SMT=2 for HPC workloads ➔ bound by FLOPs or memory bandwidth
    - A single thread executing on a core in SMT=2 will have access to **nearly all** the core's resources.
  - SMT=4 for high throughput workloads ➔ bound by front-end stalls
    - A single thread executing on a core in SMT=4 will have access to **about half** of the core's resources.

LaRC ThunderX2 Mystery SKU

LaRC ThunderX2 node (LaRC TX2), 2 sockets:

```
-cpu:0
description: CPU
product: ARM (CN9980-2200LG4077-Y21-G)
vendor: Cavium Inc.
physical id: 22
bus info: cpu@0
version: Cavium ThunderX2(R) CPU
         CN9980 v2.1 @ 2.20GHz
serial: 000011DF-020A32B7
slot: Socket 0
size: 2500MHz
capacity: 2500MHz
clock: 33MHz
capabilities: lm cpufreq
configuration: cores=32 enabledcores=28
               threads=56
```

Ordering code	Cores	Frequency (GHz)
CN9980-2500LG4077-Y21-G	32	2.5
CN9980-2400LG4077-Y21-G	32	2.4
CN9980-2300LG4077-Y21-G	32	2.3
CN9980-2200LG4077-Y21-G	32	2.2
CN9980-2100LG4077-Y21-G	32	2.1
CN9980-2000LG4077-Y21-G	32	2.0
CN9978-2300LG4077-Y21-G	30	2.3
CN9978-2200LG4077-Y21-G	30	2.2
CN9978-2100LG4077-Y21-G	30	2.1
CN9978-2000LG4077-Y21-G	30	2.0
CN9978-1800LG4077-Y21-G	30	1.8
CN9978-1800LG4077-Y21-G	30	1.8
CN9975-2400LG4077-Y21-G	28	2.4
CN9975-2300LG4077-Y21-G	28	2.3
CN9975-2200LG4077-Y21-G	28	2.2
CN9975-2100LG4077-Y21-G	28	2.1
CN9975-2000LG4077-Y21-G	28	2.0
CN9975-1800LG4077-Y21-G	28	1.8
CN9975-1800LG4077-Y21-G	28	1.8

source

STREAM Triad Results

Source	Description	GB/s
TX2 Theoretical Peak		318
arm TX2 presentation	STREAM (no TX2 SKU given)	260
Anandtech Cavium ThunderX2 9980-2200	gcc 7.2 -O2 -mcmodel=large -fopenmp -fno-PIC	241
arm colleague	“Most mid-grade TX2 SKUs will give 200-210GB/s”	200-210
LaRC TX2	gcc 8.2 -O2 -mcmodel=large -fopenmp -fno-PIC	201
NAS* Skylake Xeon Gold 6148 ×2	icc 18 -Ofast -mcmodel=large -fopenmp -fno-PIC	198

* NASA Advanced Supercomputing Division, Electra system

STREAM Observations

Source	Description	GB/s
LaRC TX2	OpenMP only, rely on first-touch to handle NUMA	190
LaRC TX2	Run with mpirun -np 2 plus OpenMP	201
LaRC TX2	gcc **-O2** -mcmodel=large -fopenmp -fno-PIC -DSTREAM_TYPE=float	135
LaRC TX2	gcc **-Ofast** -mcmodel=large -fopenmp -fno-PIC -DSTREAM_TYPE=float	200

STREAM Observations

Source	Description	GB/s
LaRC TX2	-DSTREAM_ARRAY_SIZE=84084000 -DNTIMES=100	200*
LaRC TX2	-DSTREAM_ARRAY_SIZE=699988100 -DNTIMES=20	160*

*These results average all STREAM functions

- L2 prefetcher enabled
- Best results with SMT2 mode but single thread/core, 1 MPI rank/socket
- No observable differences between gcc 8.2 and armclang 19.1
- Generally get the same BW for all STREAM functions

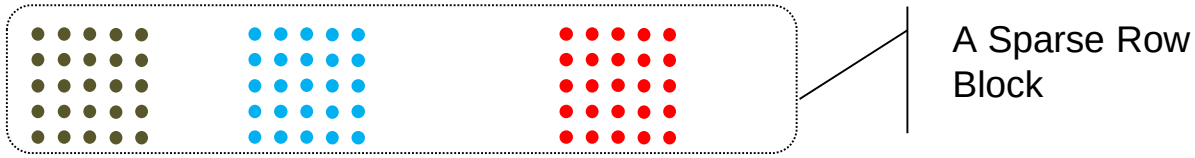
ThunderX2 Vectorization Challenges

Effective utilization of the NEON vector engine becomes challenging for block-sparse matrix-vector operations when the block size is not a multiple of the vector length.

$$\boldsymbol{Solve\ A\ \Delta q = b}$$

Algorithm 2 LINEAR SOLVER

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



Processing of a sparse row block involves multiplying a dense 5×5 block with a dense sub-vector of Δq selected based on the column index of the dense block. Repeat this computation and accumulate the resultant sub-vector of size 5.

On ThunderX2, the vector size is 128 bits. One can do up to four single precision operations. If we have a block size of four, the vectorization is straightforward.

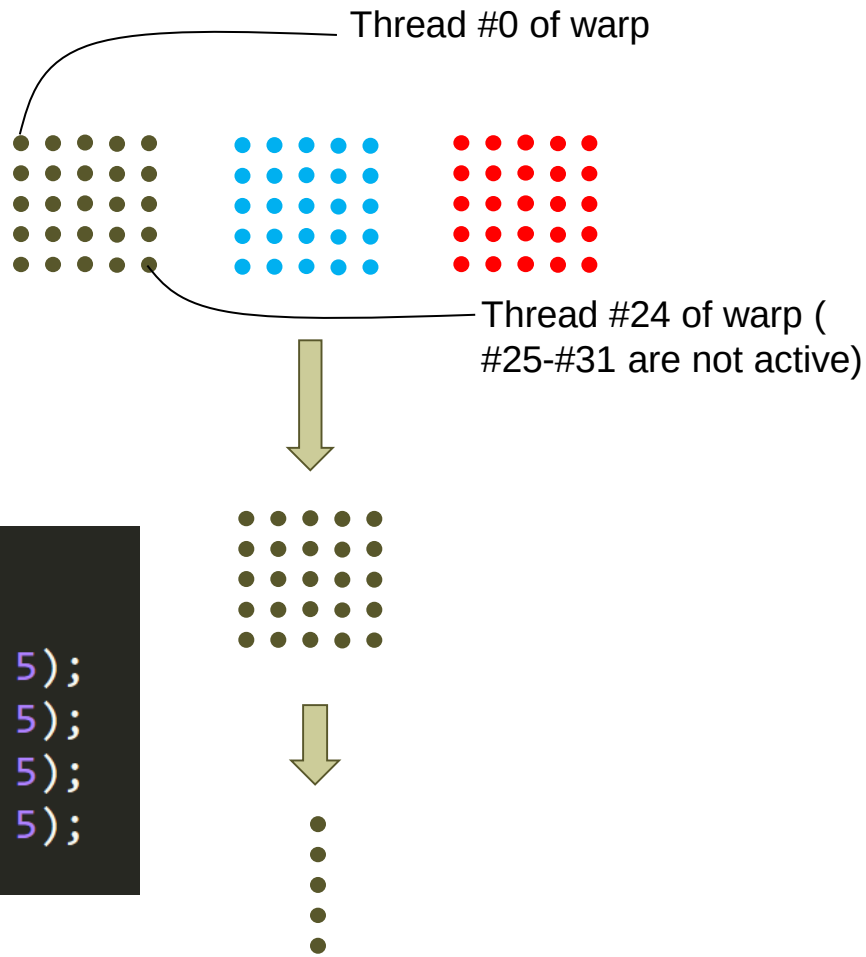
Solver Mini-app Vectorization

Let us see how we handled similar issues on GPUs and KNL.

The Single Instruction Multiple Threads (SIMT) model on GPUs works with a group of 32 threads (warp)

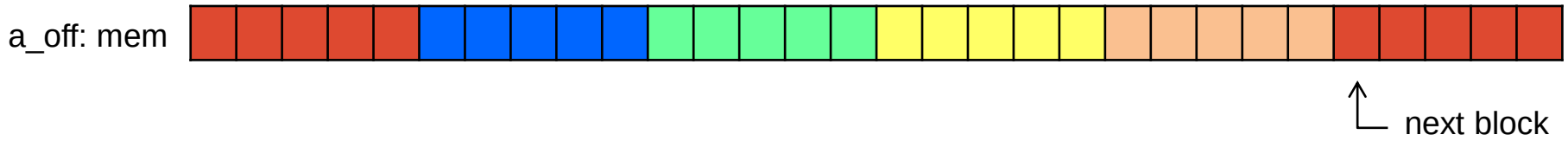
```
// Loop over Non Zeros
int j = istart - 1;
jam1 = jam[j];
for ( ; j < iend; j++) {
    jam0 = jam1;
    jam1 = jam[j + 1];
    fk += A_OFF(k, l, j) *  DQ(l, jam0 - 1);
}
```

```
// 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);
```

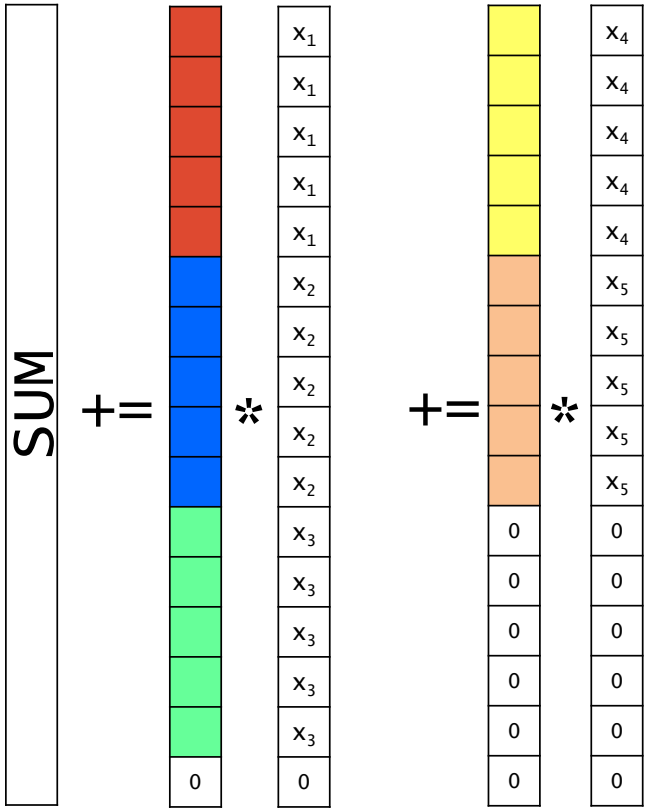


Solver Mini-app Vectorization

KNL Vector Intrinsic Implementation – AVX 512 (512 bit vector length, 16 FP32 flops)



etc



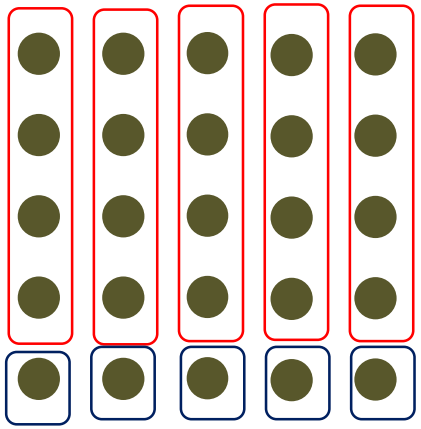
Vectorization at this level cannot be handled by the compiler.

Use of AVX-512 vector intrinsics to write low-level code

Solver Mini-app NEON Vectorization

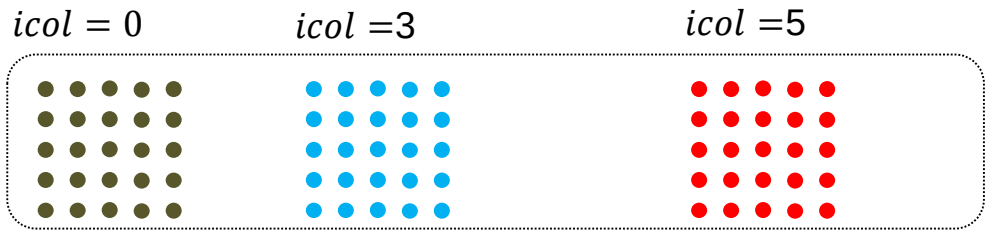
The NEON 128-bit vector can accommodate four single precision (FP32) numbers.

Experimented with several ways of vectorizing the processing of a 5x5 block.

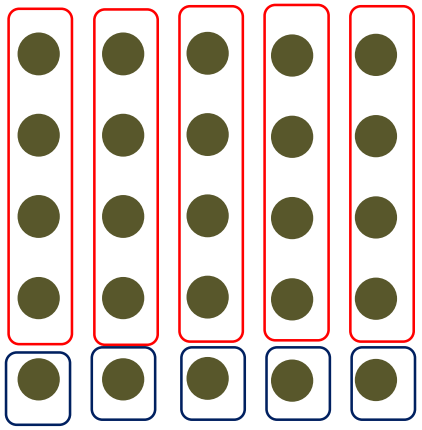


Partition a column of 5x5 block into two parts.
The first part consists of four elements that can be processed as a vector and a left-over element that can be processed as a scalar.

Solver Mini-app NEON Vectorization

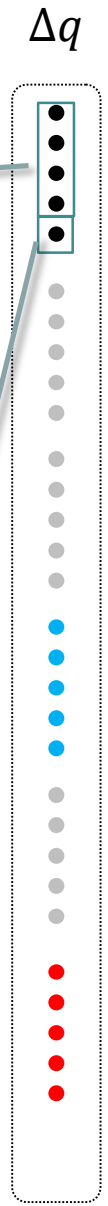


A row block is processed by an OpenMP thread one block at a time. For processing a block, load a dense block into 5 128-bit vector registers and 5 scalar registers. Output of processing a block is a column vector aggregated over the blocks.



```
a0 = vld1q_f32(a_off + j*25);
a1 = vld1q_f32(a_off + j*25 + 5);
a2 = vld1q_f32(a_off + j*25 + 10);
a3 = vld1q_f32(a_off + j*25 + 15);
a4 = vld1q_f32(a_off + j*25 + 20);
```

```
s0 = a_off[j*25+4];
s1 = a_off[j*25+9];
s2 = a_off[j*25+14];
s3 = a_off[j*25+19];
s4 = a_off[j*25+24];
```

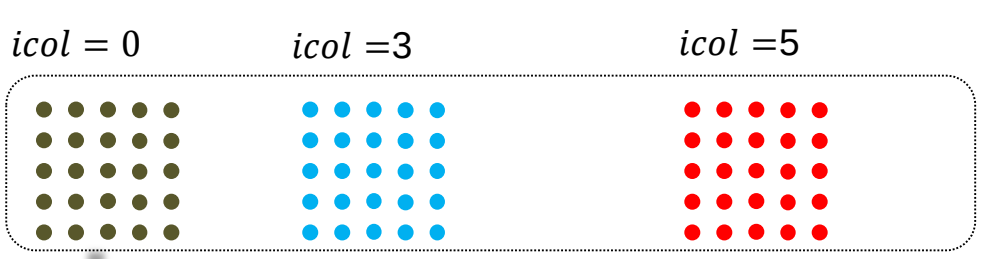


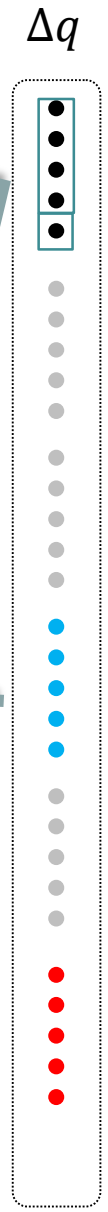

```
xv = vld1q_f32(dq + icol*5);
```

```
x0 = vgetq_lane_f32(xv, 0);
x1 = vgetq_lane_f32(xv, 1);
x2 = vgetq_lane_f32(xv, 2);
x3 = vgetq_lane_f32(xv, 3);
```

```
x4 = dq[icol*5 + 4];
```

Prefetching and Further Optimization





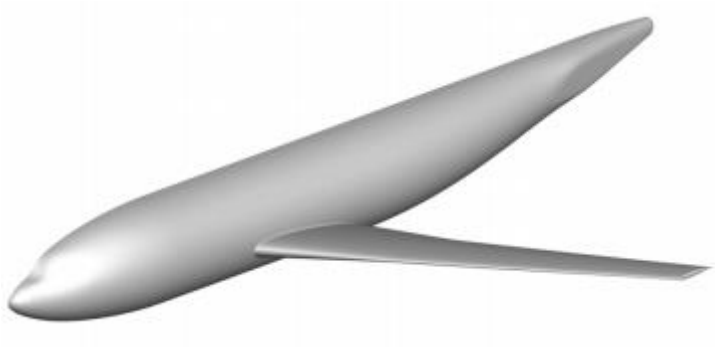
While processing the brown block, **prefetch** the blue sub-column required for processing the blue block.

```
    icol1 = jam[istart]-1;
    xv1 = vld1q_f32(dq + icol1*5);
#pragma unroll(6)
    for (j = istart; j < iend; ++j) {
      icol = icol1;
      xv = xv1;
```

We also tried intrinsic prefetching of various styles and found no performance benefit. Results for Skylake are much the same.

To decrease dependency between instructions inside the loop, we maintain a partial dense block rather than a partial column vector. Increases register pressure.

Solver Mini-app Test Case



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 solver are timed.

Architecture	Programming Model	Compiler Options
V100	CUDA	nvcc default
2× Skylake Xeon Gold 6148 (SKL)	MPI+OpenMP+C++ (w/ AVX512 intrinsics)	icpc -Ofast -mcmodel=large -qno-opt-prefetch –qopenmp -xSKYLAKE-AVX512
LaRC TX2	MPI+OpenMP+Fortran	gfortran -Ofast -mcpu=thunderx2t99 –fopenmp
LaRC TX2	MPI+OpenMP+C (w/ NEON intrinsics)	gcc -Ofast -mcpu=native -fopenmp -march=native

Solver Mini-app Results

Architecture and Prog. Model	Time (ms)	BW (GB/s)	Peak BW	%Peak
V100 (CUDA)	50	668	900	74.2
SKL (AVX512 C++)	166	202	238	84.8
LaRC TX2 (Fortran)	194	173	318	54.4
LaRC TX2 (NEON C)	167	201	318	63.2

- BW numbers for arm/Intel are computed assuming Δq is cached
- Apx. 15% speedup from intrinsics (NEON: ~3% from prefetching)
- Intrinsic speedup same for arm and Intel
- Solvers pull same BW as STREAM Triad