Optimization of a Solver for Computational Materials and Structures Problems on ARM Processor

David Wagner, James Waner
NASA Langley Research Center (LaRC)

Mohammad Zubair
Old Dominion University

Outline

- Motivation and Background
- Linear Solver
- Implementation using ARM Performance Library
- NUMA Issues
- Custom Sparse Matrix Vector Multiplication Kernel
- Conclusion

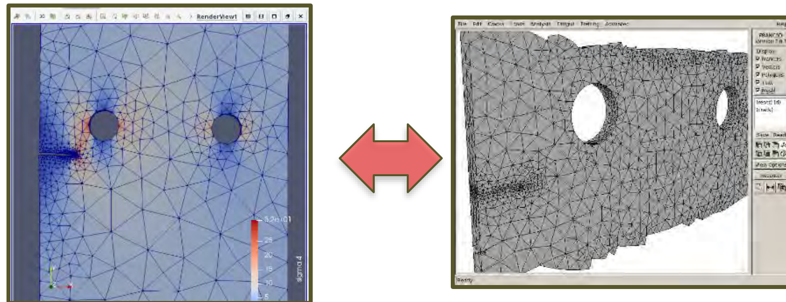
SCIFEN

(software.nasa.gov/software/LAR-18720-1)

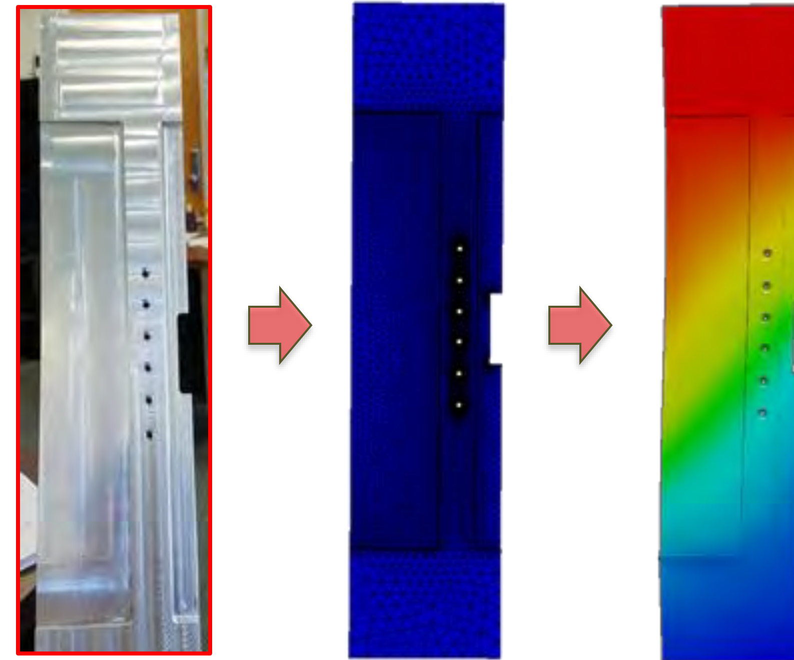
Scalable Implementation of Finite Elements by NASA

- Leverages high performance computing to solve large-scale computational materials and structures problems using the finite element method

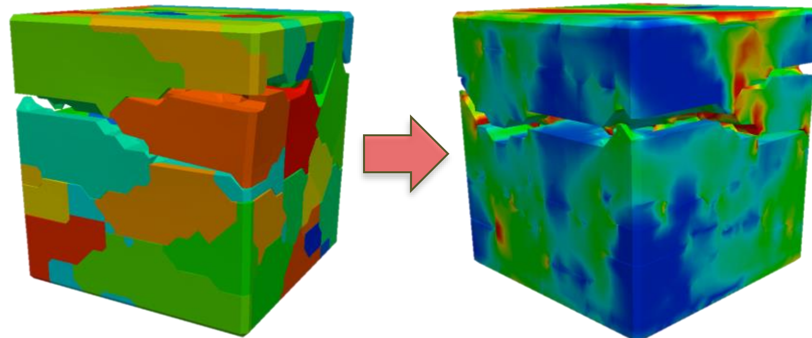
Fracture mechanics



Component-level stress analysis



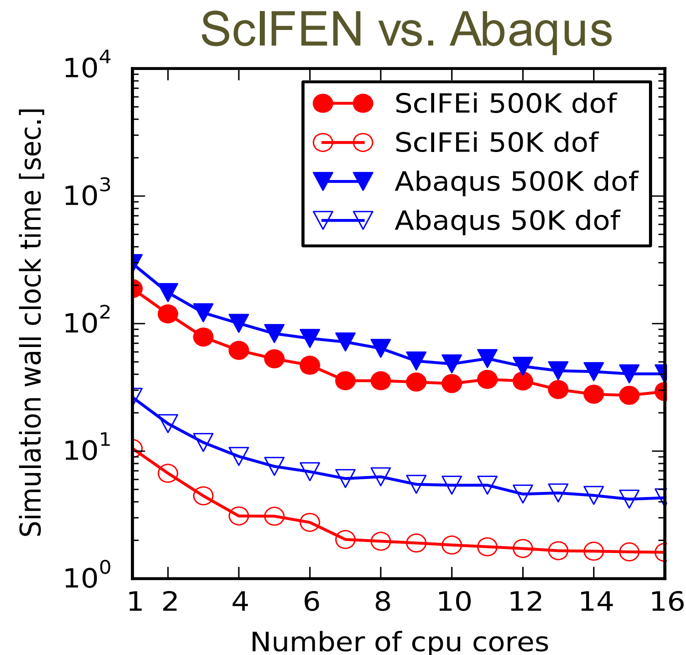
Microstructural modeling

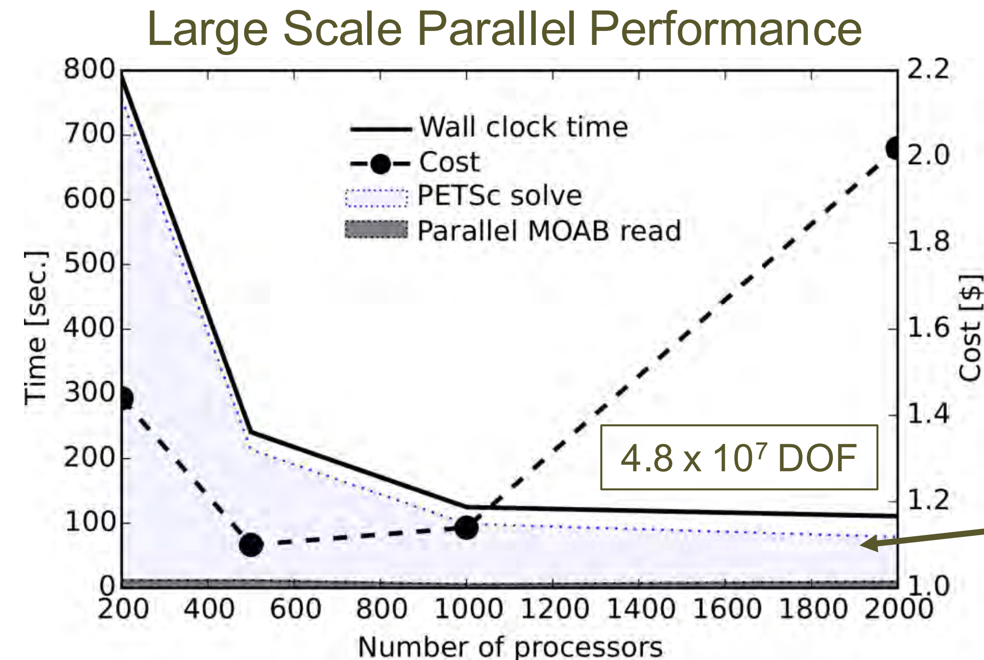


SCIFEN (software.nasa.gov/software/LAR-18720-1)

Scalable Implementation of Finite Elements by NASA

- Developed as an efficient, scalable, and **free** alternative to commercial codes like Abaqus.





Linear solve dominates overall computation time

- ScIFEN Solver Mini-App isolates the linear solver that is the computational bottleneck

SciFEN Solver Mini-App ($A\,x = b$)

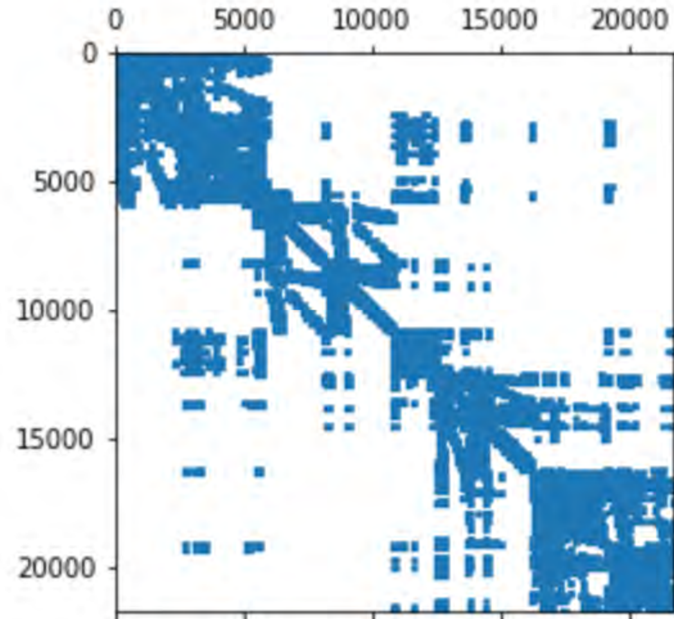
URL: https://software.nasa.gov/software/LAR-19417-1

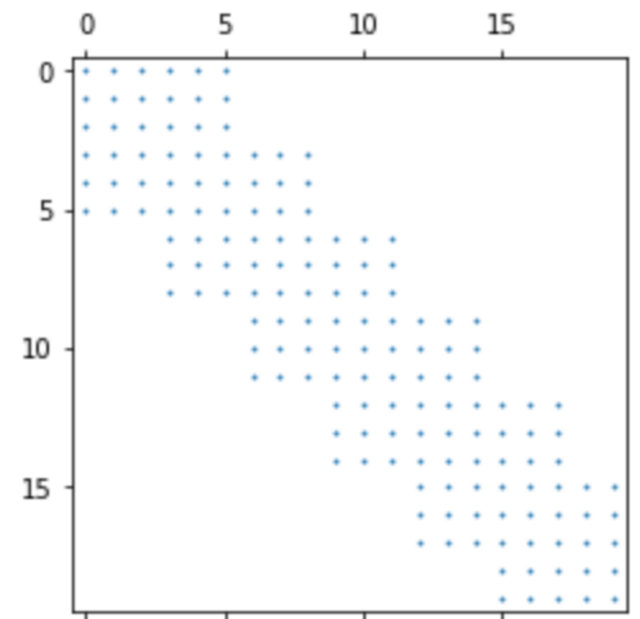
- Input: System matrix A, and a right hand side b. The matrix and vector are stored in a PETSc binary format.
- Mini-App reads input A and b using PETSc APIs
- Mini-App uses PETSc Krylov subspace iterative solvers.
- For comparison reference solution is also provided.

Linear Solver: Matrix Exploration

```python
f = open('two_hole_nocrack_mat.bin', 'rb')
param = read_data(f,4,'i',4)
nrows = param[1]
ncols = param[2]
nnz = param[3]
print(nrows, ncols, nnz)
rowsz = read_data(f,nrows,'i',4)
ja = read_data(f,nnz,'i',4)
data = read_data(f,nnz,'d',8)
f.close()
```

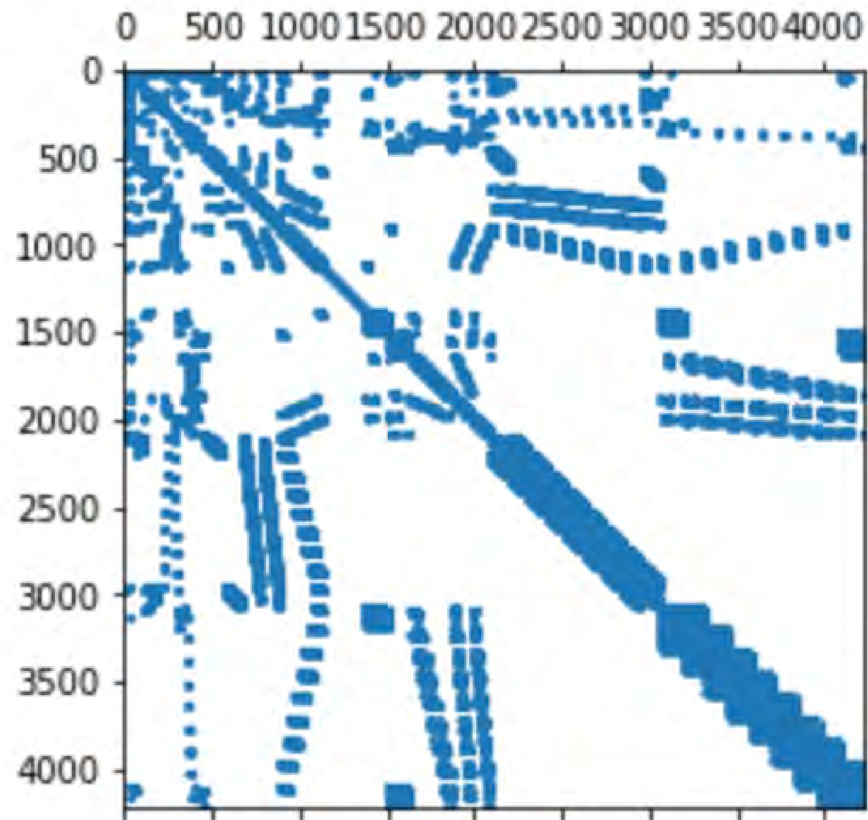
```python
plt.spy(A, marker='.', markersize=2)
plt.show()
```

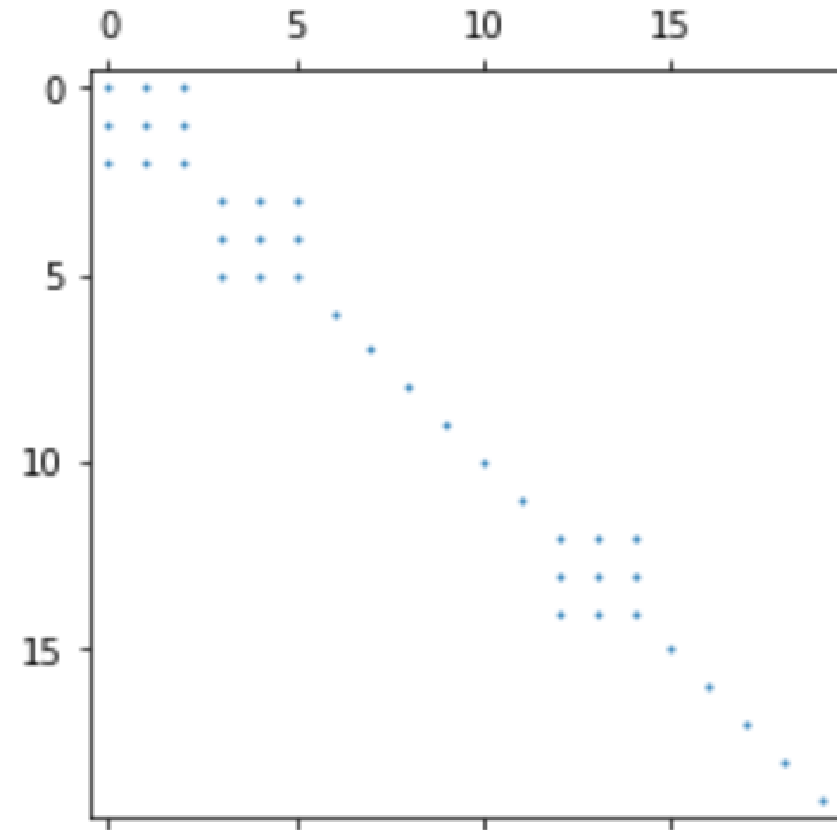




non-zero appears in blocks – this can be exploited for efficient matrix vector multiplication on HPC architectures

Small Matrix





Iterative ScIFEN Solver

Conjugate residual iterative scheme works well for a class of SciFEN applications

Algorithm 1 PRECONDITIONED CONJUGATE RESIDUAL($\mathbf{A}$, $\mathbf{M}$, $\mathbf{b}$, tol)

1: $\mathbf{d_x} = 0$
2: $\mathbf{r} = \mathbf{M} * \mathbf{b}$
3: $\mathbf{a_r} = \mathbf{Ar}$
4: $rold = \mathbf{r^t a_r}$
5: $bnorm = \sqrt{\mathbf{b^t b}}$
6: $\mathbf{p} = \mathbf{r}$
7: $\mathbf{a_p} = \mathbf{ar}$
8: $r_2 = \mathbf{r^t r}$
9: **while** $\sqrt{r_2} > tol * bnorm$ **do**
10: $\quad \mathbf{m_p} = \mathbf{M} * \mathbf{a_p}$
11: $\quad denom = \mathbf{a_p^t m_p}$
12: $\quad \alpha = rold/denom$
13: $\quad \mathbf{d_x} = \mathbf{d_x} + \alpha \mathbf{p}$
14: $\quad \mathbf{r} = \mathbf{r} - \alpha \mathbf{m_p}$
15: $\quad \mathbf{a_r} = \mathbf{Ar}$
16: $\quad rnew = \mathbf{r^t a_r}$
17: $\quad \beta = rnew/rold$
18: $\quad \mathbf{p} = \mathbf{r} + \beta \mathbf{p}$
19: $\quad \mathbf{a_p} = \mathbf{a_r} + \beta \mathbf{a_p}$
20: $\quad rold = rnew$
21: $\quad r_2 = \mathbf{r^t r}$
22: **end while**
23: **return** $\mathbf{d_x}$

ScIFEN Solver using ARM Performance Library

- ARM Performance Library : Optimized serial and parallel libraries for different CPUs including Thunder X2
    - BLAS, LAPACK, FFT, and Sparse Matrix Vector Multiplication
- Use OpenMP multi-threaded implementation of Arm Performance Libraries
    - Dot Product (*cblas_ddot*), Vector Update (*cblas_daxpy)*, Sparse Matrix Vector Multiplication (*armpl_spmv_exec_d*)

Conjugate Residual ScIFEN Solver using ARMPL

Algorithm 2 PRECONDITIONED CONJUGATE RESIDUAL($\mathbf{A}$, $\mathbf{M}$, $\mathbf{b}$, tol)

1: $\mathbf{a_r} = \mathbf{Ar}$ ($armpl_spmv_exec_d$)
2: $rold = \mathbf{r^t a_r}$($cblas_ddot$)
3: $bnorm = \sqrt{\mathbf{b^t b}}$ ($cblas_ddot$)
4: $\mathbf{p} = \mathbf{r}$ ($cblas_dcopy$)
5: $\mathbf{a_p} = \mathbf{ar}$ ($cblas_dcopy$)
6: $r_2 = \mathbf{r^t r}$ ($cblas_ddot$)
7: **while** $\sqrt{r_2} > tol * bnorm$ **do**
8: $\quad\mathbf{m_p} = \mathbf{M} * \mathbf{a_p}$($element\text{-}wise\ vector\ multiplication\text{-}custom\ kernel$)
9: $\quad denom = \mathbf{a_p^t m_p}$($cblas_ddot$)
10: $\quad\alpha = rold/denom$
11: $\quad\mathbf{d_x} = \mathbf{d_x} + \alpha\mathbf{p}$($cblas_daxpy$)
12: $\quad\mathbf{r} = \mathbf{r} - \alpha\mathbf{m_p}$($cblas_daxpy$)
13: $\quad\mathbf{a_r} = \mathbf{Ar}$ ($armpl_spmv_exec_d$)
14: $\quad rnew = \mathbf{r^t a_r}$($cblas_ddot$)
15: $\quad\beta = rnew/rold$
16: $\quad\mathbf{p} = \mathbf{r} + \beta\mathbf{p}$($cblas_dscal,\ cblas_daxpy$)
17: $\quad\mathbf{a_p} = \mathbf{a_r} + \beta\mathbf{a_p}$($cblas_dscal,\ cblas_daxpy$)
18: $\quad rold = rnew$
19: $\quad r_2 = \mathbf{r^t r}$ ($cblas_ddot$)
20: **end while**
21: **return** $\mathbf{d_x}$

Conjugate Residual ScIFEN Solver using ARMPL

7: **while** $\sqrt{r_2} > tol * bnorm$ **do**

8: $\mathbf{m_p} = \mathbf{M} * \mathbf{a_p}(custom\ kernel)$

9: $denom = \mathbf{a_p^t m_p}(cblas_ddot)$

10: $\alpha = rold/denom$

11: $\mathbf{d_x} = \mathbf{d_x} + \alpha \mathbf{p}(cblas_daxpy)$

12: $\mathbf{r} = \mathbf{r} - \alpha \mathbf{m_p}(cblas_daxpy)$

13: $\mathbf{a_r} = \mathbf{Ar}\ (armpl_spmv_exec_d)$

14: $rnew = \mathbf{r^t a_r}(cblas_ddot)$

15: $\beta = rnew/rold$

16: $\mathbf{p} = \mathbf{r} + \beta \mathbf{p}(cblas_dscal,\ cblas_daxpy)$

17: $\mathbf{a_p} = \mathbf{a_r} + \beta \mathbf{a_p}(cblas_dscal,\ cblas_daxpy)$

18: $rold = rnew$

19: $r_2 = \mathbf{r^t r}\ (cblas_ddot)$

20: **end while**

```
while (sqrt(r1) > tol*bnorm && k <= max_iter)
{
    vector_mult(adiag, d_ap, d_miap, N);
    denom = cblas_ddot(N,d_ap,1,d_miap,1);
    alpha = rar_old/denom ;
    cblas_daxpy(N, alpha, d_p, 1, d_x, 1);
    nalpha = -alpha;
    cblas_daxpy(N, nalpha, d_miap, 1, d_r, 1);
    info = armpl_spmv_exec_d(.,armpl_mat, d_r, dzero, d_ar);
    rar_new = cblas_ddot(N,d_r,1,d_ar,1);
    beta = rar_new/rar_old;
    cblas_dscal(N, beta, d_p, 1);
    cblas_daxpy(N, done, d_r, 1, d_p, 1) ;
    cblas_dscal(N, beta, d_ap, 1);
    cblas_daxpy(N, done, d_ar, 1, d_ap, 1) ;
    rar_old = rar_new;
    r1 = cblas_ddot(N, d_r, 1, d_r, 1);

}
```

Performance of CR on Thunder X2 for Test Matrices

MATRIX	Iterations to Converge	Total Time (ms)	Sparse Matrix Vector Time (ms)	% Time in Sparse Matrix Vector Oper.
100K	171	317	206	65%
250K	308	1549	1189	77%
500K	292	2837	2341	83%
1M	514	10477	8765	84%

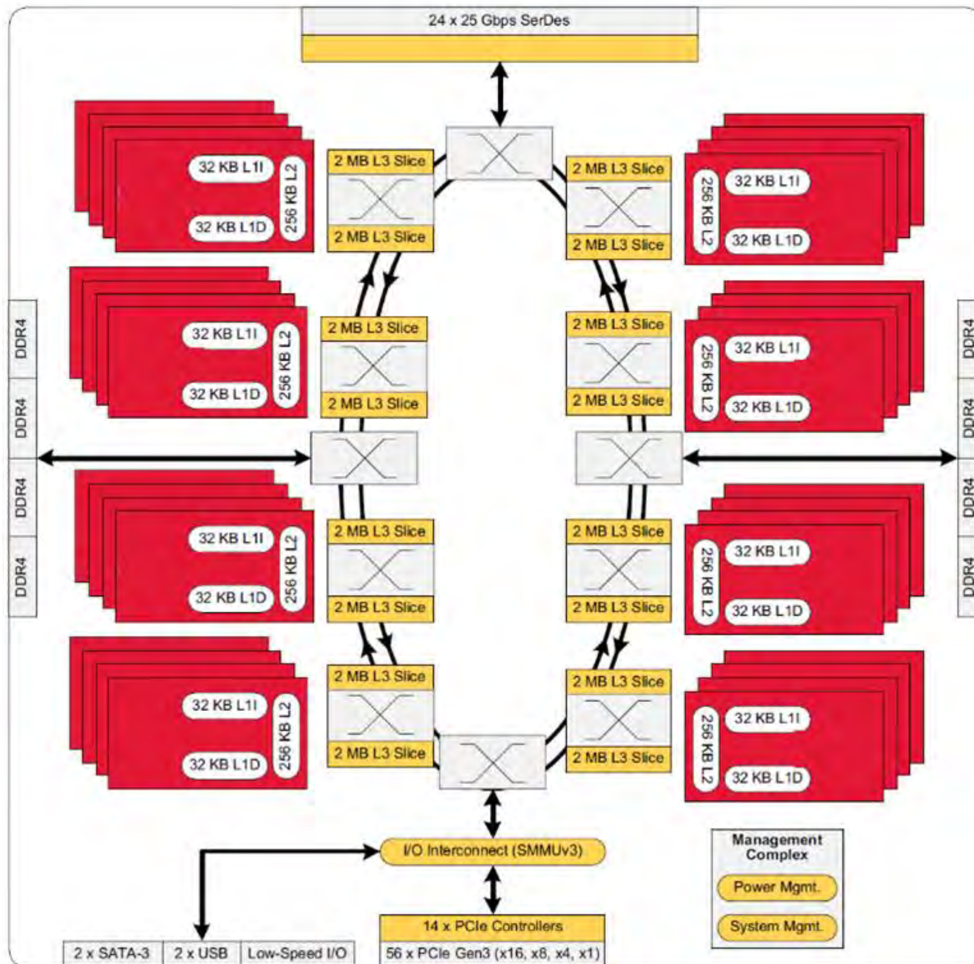
Performance Analysis of ARMPL Sparse Matrix Vector Library on Thunder X2

MATRIX	N	NNZ	BYTES	ARMPL SPMV TIME (ms)	BW (GB/s)	% OF THEOR. PEAK (260 GB/s)
100K	100704	8201934	67226736	0.7800	86	33%
250K	251733	20732877	169890744	3.5100	48	19%
500K	511167	42448329	347765304	7.8200	44	17%
1M	1070391	89131761	730180344	16.5600	44	17%

All matrices were unstructured block-sparse matrices with block size of 3×3

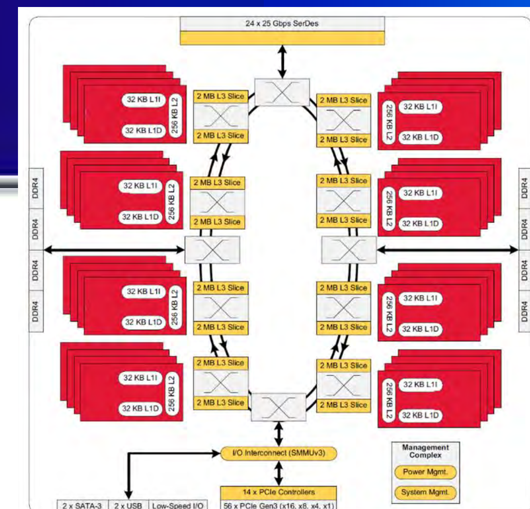
ARMPL does not support Block CSR format, we used the CSR format.

Thunder X2 Processor Overview



- Up to 32 custom Armv8.1 cores, up to 2.5GHz
- Quad issue fully out of order, 4 threads per core
- 32KB L1 I/D Cache; 256KB L2 per core
- 32 MB distributed L3 cache (1MB per core). No L3 cache affinity to cores
- The cores are interconnected using a bidirectional ring bus. Cores are organized as quad-core blocks with the L3 cache being the crossbar between the cores in the block.

Thunder X2 Processor Overview - Dual Socket



DRAM Memory Bandwidth: 159 GB/s per socket (318 GB/s for dual-socket node). Note that the memory attached to a Thunder X2 is local (faster access) for threads running on that processor.

ThunderX2 Performance Overview

- **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**
  - Up to 130GB/s per socket (i.e. 260GB for dual-socket node) for STREAM Triad*
- **Hardware prefetching into L2**
- **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 **<u>nearly all</u>** 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 **<u>about half</u>** of the core's resources.

* provided by ARM

Thunder X2 Performance Challenge

Memory: STREAM Bandwidth		
Mem Hierarchy	**Compiler & OS settings**	**Result**
Cavium ThunderX2 Gcc 7.2 binary	-O2 -mcmodel=large -fopenmp -DVERBOSE -fno-PIC" OMP_PROC_BIND=spread	241 GB/s
Cavium ThunderX2 Gcc 7.2 binary	*-Ofast -fopenmp -static* OMP_PROC_BIND=spread	157 GB/s
Cavium ThunderX2 Gcc 7.2 binary	OMP_PROC_BIND not configured	118 GB/s
Intel ICC Binary	*-fast -qopenmp -parallel* KMP_AFFINITY=verbose,scatter	183 GB/s
Intel gcc Binary	*Ofast -fopenmp -static* OMP_PROC_BIND=spread	151 GB/s
Intel gcc Binary	*Ofast -fopenmp -static* OMP_PROC_BIND not configured	150 GB/s

https://www.anandtech.com/show/12694/assessing-cavium-thunderx2-arm-server-reality/6

"Theoretically, the ThunderX2 has 33% more bandwidth available than an Intel Xeon, as the SoC has 8 memory channels compared to Intel's six channels. These high bandwidth numbers can only be achieved in very specific conditions and require quite a bit of tuning to avoid reaching out to remote memory."

The main computation for the linear solver is the block sparse matrix vector operation, which is memory bound operation.

To maximize performance for this operation it is essential to effectively utilize the memory bandwidth available.

This becomes challenging on a dual socket machine because of NUMA issues.

Thunder X2 – Addressing NUMA Issue

lscpu --extended --all

CPU	NODE	SOCKET	CORE	L1d:L1i:L2:L3	ONLINE	MAXMHZ	MINMHZ
0	0	0	0	0:0:0:0	yes	2500.0000	1000.0000
1	0	0	1	1:1:1:0	yes	2500.0000	1000.0000
2	0	0	2	2:2:2:0	yes	2500.0000	1000.0000
3	0	0	3	3:3:3:0	yes	2500.0000	1000.0000
4	0	0	4	4:4:4:0	yes	2500.0000	1000.0000
5	0	0	5	5:5:5:0	yes	2500.0000	1000.0000
6	0	0	6	6:6:6:0	yes	2500.0000	1000.0000
7	0	0	7	7:7:7:0	yes	2500.0000	1000.0000
8	0	0	8	8:8:8:0	yes	2500.0000	1000.0000
9	0	0	9	9:9:9:0	yes	2500.0000	1000.0000
10	0	0	10	10:10:10:0	yes	2500.0000	1000.0000
11	0	0	11	11:11:11:0	yes	2500.0000	1000.0000
12	0	0	12	12:12:12:0	yes	2500.0000	1000.0000
13	0	0	13	13:13:13:0	yes	2500.0000	1000.0000
14	0	0	14	14:14:14:0	yes	2500.0000	1000.0000
15	0	0	15	15:15:15:0	yes	2500.0000	1000.0000
16	0	0	16	16:16:16:0	yes	2500.0000	1000.0000
17	0	0	17	17:17:17:0	yes	2500.0000	1000.0000
18	0	0	18	18:18:18:0	yes	2500.0000	1000.0000
19	0	0	19	19:19:19:0	yes	2500.0000	1000.0000
20	0	0	20	20:20:20:0	yes	2500.0000	1000.0000
21	0	0	21	21:21:21:0	yes	2500.0000	1000.0000
22	0	0	22	22:22:22:0	yes	2500.0000	1000.0000
23	0	0	23	23:23:23:0	yes	2500.0000	1000.0000
24	0	0	24	24:24:24:0	yes	2500.0000	1000.0000
25	0	0	25	25:25:25:0	yes	2500.0000	1000.0000
26	0	0	26	26:26:26:0	yes	2500.0000	1000.0000
27	0	0	27	27:27:27:0	yes	2500.0000	1000.0000
28	0	0	0	0:0:0:0	yes	2500.0000	1000.0000
29	0	0	1	1:1:1:0	yes	2500.0000	1000.0000
30	0	0	2	2:2:2:0	yes	2500.0000	1000.0000
31	0	0	3	3:3:3:0	yes	2500.0000	1000.0000
32	0	0	4	4:4:4:0	yes	2500.0000	1000.0000
33	0	0	5	5:5:5:0	yes	2500.0000	1000.0000
34	0	0	6	6:6:6:0	yes	2500.0000	1000.0000
35	0	0	7	7:7:7:0	yes	2500.0000	1000.0000

CN 9975 Thunder X2 @NASA Langley
SIMT=2 Mode, Dual Socket, 28 Cores/Socket
Total Hardware Threads (CPUs): 112

Socket #0 (Numa node#0): 0-27 Cores
Socket #1 (Numa node#1): 28-56 Cores

Core 0: Hardware Threads 0, 28
Core 1: Hardware Threads 1, 29
.
.
Core 27: Hardware Threads 27, 55
Core 28: Hardware Threads 56, 84
.
.
Core 55: Hardware Threads 83, 111

Objective: For an OpenMP implementation, map OpenMP threads to hardware threads such that most of the memory accesses by a thread running on a NUMA node are from the local memory.

NUMA Issues

Code running on all 56 cores

```
-bash-4.2$ ./blasmatvec  mat_1M.bin
number of rows: 1070391
number of non zeros: 89131761
Warming....
Timing....
Total timein ms:     16.36
gold checksum:  1.06652824e+04
```

Code running on 28 cores on the same NUMA node

```
-bash-4.2$ numactl --physcpubind=0-27  ./blasmatvec  mat_1M.bin
number of rows: 1070391
number of non zeros: 89131761
Warming....
Timing....
Total timein ms:     10.78
gold checksum:  1.06652824e+04
```

NUMA Issues

- Handling NUMA issues requires that every thread running on a NUMA node access its local memory only. Handling this in an OpenMP environment at the library level can be challenging.
- Handling at the application level can by relatively easy.
  - Use MPI+OpenMP. Partition the unstructured grid in two partitions and run two MPI ranks on a Thunder X2 with one rank working with one partition mapped to a NUMA node (socket) of 28 cores.
  - This ensures that all threads of a rank access local memory.
- At the library level, depending on the problem, it is possible to make a copy of main data arrays which is NUMA friendly (use of first touch policy)

NUMA Friendly Custom Sparse Matrix Vector on Thunder X2

Use first touch policy to distribute the matrix among NUMA nodes

```
for (n = 0; n < nrows; ++n) {
  iamn[n] = iam[n];
  iamn[n+1]  = iam[n+1];
  istart = iam[n];
  iend   = iam[n+1];
  for (j = istart; j < iend; ++j) {
      icol = jam[j];
      jamn[j] = icol;

      amatvn[j*9+3*0+0] = amatv[j*9+3*0+0];
      amatvn[j*9+3*1+0] = amatv[j*9+3*1+0];
      amatvn[j*9+3*2+0] = amatv[j*9+3*2+0];


      amatvn[j*9+3*0+1] = amatv[j*9+3*0+1];
      amatvn[j*9+3*1+1] = amatv[j*9+3*1+1];
      amatvn[j*9+3*2+1] = amatv[j*9+3*2+1];
      amatvn[j*9+3*0+2] = amatv[j*9+3*0+2];
      amatvn[j*9+3*1+2] = amatv[j*9+3*1+2];
      amatvn[j*9+3*2+2] = amatv[j*9+3*2+2];
  }
} // end parallel loop
```

```
for (n = 0; n < nrows; ++n) {
  f1 = 0.0; f2 = 0.0; f3 = 0.0;
  istart = iamn[n];
  iend   = iamn[n+1];
  for (j = istart; j < iend; ++j) {
      icol = jamn[j];
      f1 = f1 + amatvn[j*9+3*0+0]*dx[icol*3+0];
      f2 = f2 + amatvn[j*9+3*1+0]*dx[icol*3+0];
      f3 = f3 + amatvn[j*9+3*2+0]*dx[icol*3+0];

      f1 = f1 + amatvn[j*9+3*0+1]*dx[icol*3+1];
      f2 = f2 + amatvn[j*9+3*1+1]*dx[icol*3+1];
      f3 = f3 + amatvn[j*9+3*2+1]*dx[icol*3+1];

      f1 = f1 + amatvn[j*9+3*0+2]*dx[icol*3+2];
      f2 = f2 + amatvn[j*9+3*1+2]*dx[icol*3+2];
      f3 = f3 + amatvn[j*9+3*2+2]*dx[icol*3+2];
  } // end istart loop

  dy[n*3+2] = f3;
  dy[n*3+1] = f2;
  dy[n*3+0] = f1;
} // end parallel loop
```

Performance of NUMA Friendly Custom Sparse Matrix Vector on Thunder X2

```
export OMP_SCHEDULE=static
export OMP_NUM_THREADS=56
export OMP_PROC_BIND=true
export OMP_PLACES='{0}:28:1,{56}:28:1'
OMP_PLACES = '{0},{1},{2},{3},{4},{5},{6},{7},{8},{9},{10},{11},{12},{13},{14},{15},{16},{17},{18},{19},{20},{21},{22},{23},{24},{25},{26},{27},
{56},{57},{58},{59},{60},{61},{62},{63},{64},{65},{66},{67},{68},{69},{70},{71},{72},{73},{74},{75},{76},{77},{78},{79},{80},{81},{82},{83}'
```

MATRIX	N	NNZ	BYTES	TIME (ms) ARMPL SPMV	TIME (ms) CUSTOM	BW (GB/s) ARMPL SPMV	BW (GB/s) CUSTOM SPMV
100K	100704	8201934	67226736	0.7800	0.2800	86	240
250K	251733	20732877	169890744	3.5100	0.7100	48	239
500K	511167	42448329	347765304	7.8200	1.5500	44	224
1M	1070391	89131761	730180344	16.5600	3.6400	44	201

Execution time is based on the minimum across multiple runs. Note that for small matrices the caching effect results in a better performance and higher bandwidth.

Performance of Updated ARMPL on Thunder X2

N	ARMPL 19.3 (ms)	ARMPL 20 BSR rm (ms)	ARMPL 20 BSR cm (ms)	Custom (ms)
100K	0.32258	0.21700	0.21790	0.28000
250K	1.24070	0.84850	0.76900	0.71000
500K	3.40725	1.96370	1.94190	1.55000
1M	7.82124	4.76400	5.23270	3.64000

Courtesy: Christopher Armstrong and Keeran Brabazon

Conclusion

- ARM processor looks promising platform for SciFEN application
- Demonstrated that near optimal performance is possible for Sparse Matrix Vector Operation on Thunder X2
- Addressing NUMA issues is critical to achieve optimal performance