Sequences of Sparse Matrix-Vector Multiplication on Fugaku's A64FX processors

EAHPC 2021

Jérôme Gurhem [1,*], Maxence Vandromme [3], Miwako Tsuji [4], Serge G. Petiton [2,3], Mitsuhisa Sato [4]

1. Aneo, Boulogne-Billancourt, France
*(3) during the work
2. UMR 9189 - CRIStAL, Univ. Lille, CNRS, Lille, France
3. USR 3441 - Maison de la Simulation, CNRS, Saclay, France
4. RIKEN Center for Computational Science, Kobe, Japan

September 7th 2021

Section 1

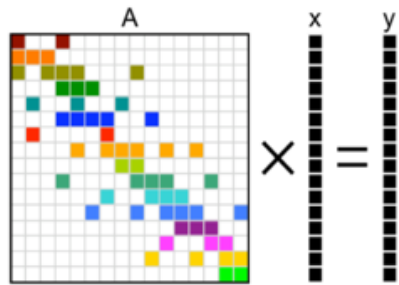
Introduction

Introduction

A.x

- Sparse matrix vector product A.x
- Very common in scientific computation applications

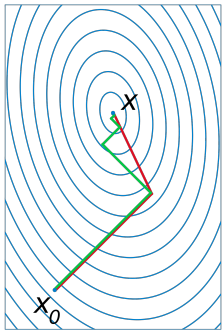
A.x



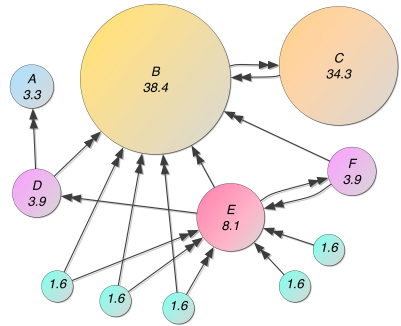
Introduction

A.x = building block for more complex algorithms

Conjugate gradient



PageRank



⇒ sequences of A.x operations with vector additions
⇒ A(A.x + x) + x as proxy application for iterative methods

Introduction

Goals

- Compare the performances between A.x and A(A.x + x) + x
- Use matrices deviating more from the diagonal
- Use MPI and OpenMP to scale on more than 1 node
- Evaluate the performance gain when using SVE for this application

Section 2

Methods and Data

Matrix storage formats

Four storage formats used

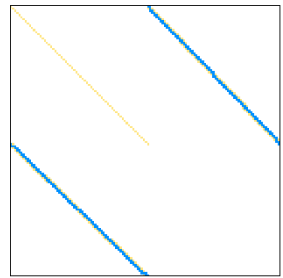
- CSR and ELLPACK : row-major formats
- COO : coordinates
- SCOO : ordered coordinates to allow a 2D block distribution

About this choice

- CSR and ELLPACK are commonly used in libraries/software/packages
- COO and SCOO use different storage methods and show different performances patterns
- They are also used in AI libraries such as TensorFlow, Pytorch, ...
- Many more optimized storage formats have been proposed for SpMV
- However, we want to study the performance that can be expected with applications using regular storage formats

Data

nlpkkt120



nlpkkt120

- Diagonal matrix with two bands
- Representative of matrices typically used as benchmarks
- e.g. HPCG also uses band diagonal matrices
- Comparison with previous study on SpMV for A64FX

Data

Matrix Market

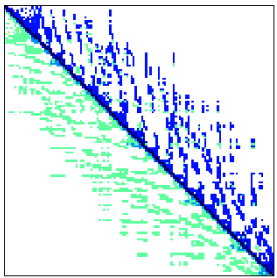
- Two other matrices from Matrix Market
- More sparse than *nlpkkt120*
- And with more deviation from the diagonal

Matrix Market

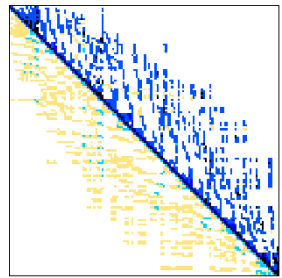
matrix	rows	columns	nnz
cage14	1,505,785	1,505,785	27,130,349
cage15	5,154,859	5,154,859	99,199,551
nlpkkt120	3,542,400	3,542,400	96,845,792

Data

cage14



cage15

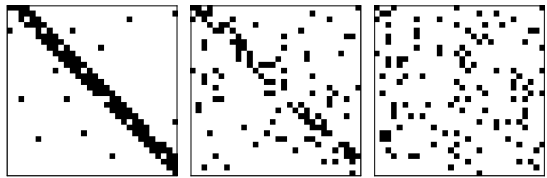


Data

C-diagonal Q-perturbed matrices

- Starting with C values per row and column
- Probability of Q to change the column index of the value
- Q small = most of the values near the diagonal
- Q large = dispersion of the values in the matrix
- Parameter variation in experiments

$C = 0.4 ; Q = \{0.05, 0.5, 0.9\}$



Environment

Computing environment and parameters

- 1 or 2 A64FX processors on the Fugaku supercomputer
- Fujitsu compiler in Clang mode
- -Kopenmp -fPIC -Ofast -mcpu=native -funroll-loops -fno-builtin -march=armv8.2-a+sve
- Reporting the best performances, in GFlop/s
- With OpenMP+MPI, the #threads per MPI processes changes, but the #threads total remains constant

Section 3

Results

nlpkkt120 - A.x application

MPI+OpenMP on 1 and 2 processors, varying #threads per MPI process (GFlop/s)

threads	COO		CSR		ELL		SCOO	
	1	2	1	2	1	2	1	2
1	7.9	10.0	75.6	127.2	81.7	145.5	6.0	11.9
2	3.8	4.4	75.0	136.4	72.4	135.0	6.0	11.9
4	5.4	4.0	73.6	142.2	80.3	135.0	5.9	11.7
6	5.9	2.6	73.8	130.0	64.2	131.7	5.8	11.4
12	3.1	3.1	72.5	131.5	69.9	129.4	5.5	11.0
24	3.0	2.4	71.9	130.9	69.1	124.5	3.2	5.4
48	1.6	2.6	69.8	127.9	68.8	123.2	2.4	3.3
MPI	14.5	27.4	78.1	119.1	84.4	121.2	6.8	13.0

Observations

- COO and SCOO perform very poorly
- Good scaling on 2 nodes with CSR and ELLPACK; coherent with results from previous study

nlpkkt120 - A(A.x + x) + x application

MPI+OpenMP on 1 and 2 processors, varying #threads per MPI process (GFlop/s)

threads	COO		CSR		ELL		SCOO	
	1	2	1	2	1	2	1	2
1	2.6	1.1	8.9	10.4	8.3	10.6	3.8	5.9
2	2.8	1.7	8.2	9.2	8.1	9.1	3.7	5.9
4	3.2	3.1	13.1	15.0	13.9	14.8	4.3	7.2
6	2.3	1.9	16.7	17.6	16.0	17.9	4.6	7.5
12	2.1	1.9	18.1	20.8	17.2	22.5	4.6	8.0
24	3.6	1.8	27.5	25.9	29.6	25.1	3.6	5.6
48	1.6	3.4	34.1	35.8	33.4	36.1	3.3	4.3
MPI	3.2	1.2	9.0	9.9	8.1	9.9	3.6	5.6

Observations

- COO and SCOO still perform very poorly
- Much worse for CSR and ELLPACK compared to A.x; no scaling on 2 nodes; gather result vector

All three Matrix Market matrices

1 processor, A.x application (GFlop/s)

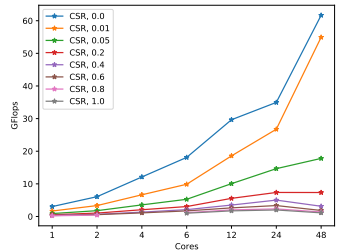
matrixtype	CPPOMP		MPI		MPIOMP	
	CSR	ELL	CSR	ELL	CSR	ELL
cage14	29.5	23.1	31.7	24.7	32.4	25.2
cage15	28.8	22.2	31.0	24.5	33.2	26.2
nlpkkt120	78.9	81.7	78.1	84.4	75.6	81.7

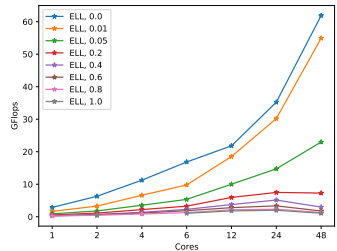
1 processor, A(A.x + x) + x application (GFlop/s)

matrixtype	CPPOMP		MPI		MPIOMP	
	CSR	ELL	CSR	ELL	CSR	ELL
cage14	27.9	22.0	5.1	4.2	15.3	14.0
cage15	26.0	20.5	5.8	5.9	17.1	14.7
nlpkkt120	62.6	60.6	9.0	8.1	34.1	33.4

C-diagonal Q-perturbed, OpenMP on 1 processor, CSR/ELL, C = 100, varying Q

$A(A.x + x) + x$



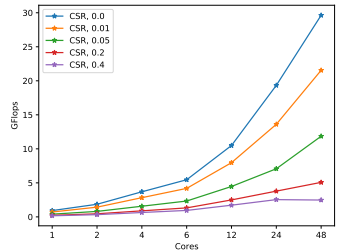


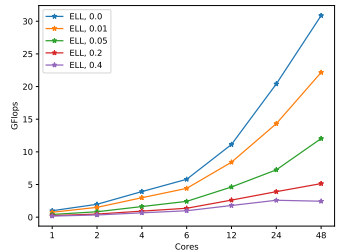
Observations

- Performance very dependent on the Q parameter
- i.e. the degree of deviation from the diagonal
- More cache misses when the elements are more spread out
- Even relatively small deviations (Q = 0.05) have a large impact

C-diagonal Q-perturbed, OpenMP on 1 processor, CSR/ELL, C = 8, varying Q

$A(A.x + x) + x$





Observations

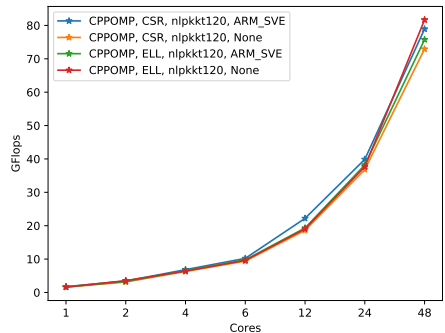
- Changing C from 100 to 8 non-zero elements per line
- Same performance patterns overall
- About 2x slower than for C = 100
- So the sparsity also has an impact on the performance

Scalable Vector Extension (SVE) for ARM

SpMV kernel - SVE code

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < this->ln_row; i++) {
  svfloat64_t tmp; tmp = svadd_z(svpfalse(), tmp, tmp);
  int start = this->rowptr[i];
  int end = this->rowptr[i + 1];
  int j = start;
  uint64_t fix = this->f_col;
  svbool_t pg = svwhilelt_b64(j, end);
  do {
    svfloat64_t values_vec = svld1(pg, &(this->values[j]));
    svuint64_t col = svld1sw_u64(pg, &(this->colidx[j]));
    svuint64_t col_fix = svsub_z(pg, col, fix);
    svfloat64_t v_vec = svld1_gather_index(pg, v, col_fix);
    tmp = svmla_m(pg, tmp, values_vec, v_vec);
    j += svcntd();
    pg = svwhilelt_b64(j, end);
  } while (svptest_any(svptrue_b64(), pg));
  r[i] = svaddv(svptrue_b64(), tmp);
}
```

With and without SVE on A.x, CSR/ELL



Section 4

Conclusion

Conclusion and Perspectives

Main observations

- The sequence of matrix vector products A(A.x + x) + x is harder than A.x
- Good scaling on more than one processor in the simple cases (diagonal) with A.x
- Performance drops quickly with deviation from the diagonal
- A naive implementation using SVE intrinsics did not improve performance

Conclusion and Perspectives

Perspectives

- Study the application behaviour to understand SVE more: profiling, compilers...
- Scale to more than 2 processors and larger data sets (e.g. graph data for PageRank)
- Implement and evaluate conjugate gradient

Thank you for your attention !