High bandwidth memory architectures for very sparse systems

Frank Hülsemann[1]

[1]EDF R&D

SparseDays 2019, CERFACS
July 2019

Overview

→ Context: Computational PDEs at EDF

→ The roofline model

→ Example of a HMB architecture: NEC SX-Aurora Tsubasa

→ Computational intensity at the example of the Conjugate Gradient

→ Performance predictions and results (mono-core)

→ Outlook

Computational PDEs at EDF

EDF is an energy utility company

→ with a certain number of computational applications (PDEs or not);

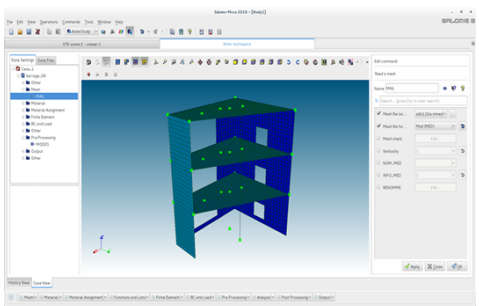
→ with more than 30 years of experience in the development of in-house codes;

→ that is represented in the Top500 since Nov. 2006 (EDF or EDF R&D).

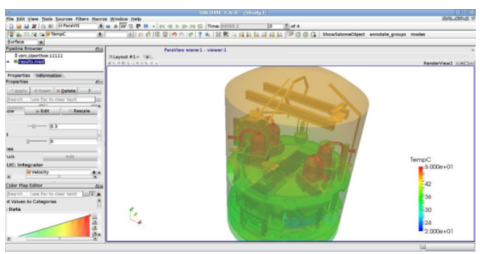
Common platform for computational PDEs: Salome

EDF R&D develops and distributes several specializations of the Salome platform for different application areas:

Salome-Meca

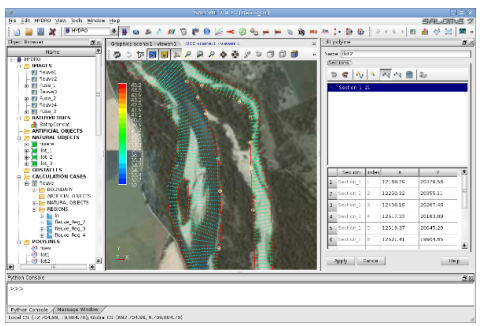


Salome_CFD



Salome_CFD

Salome-Hydro



For more information, see https://www.salome-platform.org/contributions/edf_products.

Computational PDEs at EDF in general

Computational PDEs:

→ Discretizations:

- (classical) FE, FV
- more recently: robust discretizations, such as Compatible Discrete Operators (CDO), Hybrid High Order methods (HHO)
- in general with compact support

→ Meshes:

- in general unstructured, with notable exceptions
- from tens of thousands to more than a billion cells (CFD in 2017)

→ Types of computations:

- rarely one-shot
- general case: time-dependent, non-linear, parametric

Single core performance matters!

. . . and in CFD in particular

Discretizations with compact support on unstructured meshes
$\Rightarrow$ sparse matrices without any particular pattern.

- $\rightarrow$ co-localized FV: Entries per row = number of faces + 1 (per variable, modulo boundary conditions)
  - on tetrahedra: 5 non-zeros per row
  - on hexahedra: 7 non-zeros per row
- $\rightarrow$ CDO face-based or HHO(k=0) after static condensation: As many entries per row as faces in the two cells that share a given face.
  - on tetrahedra: 7
  - on hexahedra: 11

In other words: In general, **very sparse** unstructured systems.

The importance of the memory bandwidth

The *roofline model* of Williams, Waterman and Patterson (2009) states that

$$P\,[F/s] = \text{min(comp. intensity [F/B] * bandwidth [B/s], peak perf. [F/s])}$$

is an upper bound for the (floating-point) performance of an algorithm. Here,

$$\text{computational intensity [F/B]} = \frac{\text{number of arithmetic operations [F]}}{\text{number of bytes read or written [B]}}$$

On a given architecture, the performance that an algorithm can obtain is limited by the peak performance, if

$$\text{comp. intensity [F/B]} \geq \frac{\text{peak perf. [F/s]}}{\text{bandwidth [B/s].}}$$

Otherwise, the performance of the algorithm is limited by the available memory bandwidth.

Peak performance and memory bandwidth: Examples

Intel Xeon Gold 6140 (2.3 GHz, AVX-512, dual socket, 18 cores per socket):

peak performance per core:	73.6 GF/s (according to Top500)
memory bandwidth for 1 core:	**12.6 GB/s** (Stream benchmark)
max. memory bandwidth	163 GB/s (Stream benchmark, 36c)

$\Rightarrow$ On one such core, an algorithm with a computational intensity less than **5.8 [F/B]** is limited by the memory bandwidth.

NEC SX-Aurora Tsubasa 10B (1.4 GHz, 8 cores):

peak performance per core:	268.8 GF/s (according to NEC)
memory bandwidth for 1 core:	**350 GB/s** (Stream benchmark)
max. memory bandwidth	977 GB/s (Stream benchmark, 4c)

On this machine, if comp. intensity $<$ **0.77 [F/B]**, then memory bandwidth limited.

NEC SX-Aurora Tsubasa

NEC SX-Aurora Tsubasa 10B:

- → Vector computer on a PCI-Express card
- → 8 vector cores, 48 GB HBM2 memory, 16MB cache per card
- → Std. execution model: Whole program runs on the vector card.
- → Big/little approach: Each vector core (big) contains a scalar CPU (little).
- → Vector length: 256 (double precision values, 64bit)
- → Programming environment: vectorizing compilers for Fortran, C, C++; MPI, OpenMP
- → Cross compilation on x86 host.

The conjugate gradient as solver example

Only few building blocks necessary ($\alpha \in \mathbb{R}$, $x, y \in \mathbb{R}^N$, $A \in \mathbb{R}^{N \times N}$):

daxpy	$y \leftarrow \alpha x + y$
inner product	$< x, y >= \sum_i x_i y_i$
vector norm	$\|x\| = \sqrt{< x, x >}$
daypx	$y \leftarrow \alpha y + x$
spmv	$y = Ax$

Note: Non-trivial preconditioners are a topic for the future.

Computational intensity of the CG building blocks

Setting: 8B double, 4B int, Ellpack storage, E=7 entries per row

Operation	**comp. intensity** [F/B]
daxpy	$2/24 = 1/12$
inner product	$1/16 \leq \text{ci} \leq 1/8$ (depends on the accumulation)
vector norm	?? (depends on the square root algorithm)
daypx	$2/24 = 1/12$
spmv	$(2\text{E-}1)/21\text{E} = 13/148 \approx 0.088$

All operations are memory bandwidth limited on both architectures (NEC: ci$<$ 0.77, Intel: ci $<$ 5.8 [F/B]).

Performance: predicted ...

Setting: 8B double, 4B int, Ellpack storage, E=7 entries per row, NEC SX-Aurora T10B

Operation	**comp. intensity** [F/B]	**performance** [GF/s] **predicted**
daxpy	1/12	(1/12)*350 = 29,2
inner product	$1/16 \leq ci \leq 1/8$	350/16 = 21,9, 350/8= 43,8
vector norm	??	??
daypx	1/12	29,2
spmv	13/148	30,7

Performance: . . . and measured

Setting: 8B double, 4B int, Ellpack storage, E=7 entries per row, NEC SX-Aurora T10B, N=309248

Operation	**comp. intensity** [F/B]	**performance** [GF/s] **predicted**	**measured**
daxpy	1/12	29,2	28,9
inner product	$1/16 \leq \text{ci} \leq 1/8$	21,9 - 43,8	34,8
vector norm	??	??	59,4
daypx	1/12	29,2	30,1
spmv	13/148	30,7	**17,3**

Performance: Some remarks

→ Getting optimal performance on vector operations, including the inner product, is **simple**. Just write down the loop and the compiler does the rest.

→ The matrix-vector product does not get close to the upper bound. The best results were obtained with an in-house version, specialised for 7 non-zeros per row with redundant computations at the boundary:

```
do ri = 1, nrows
   vr(ri) = matv(ri,1)*vi(colind(ri,1)) &
          + matv(ri,2)*vi(colind(ri,2)) &
          + matv(ri,3)*vi(colind(ri,3)) &
          + matv(ri,4)*vi(colind(ri,4)) &
          + matv(ri,5)*vi(colind(ri,5)) &
          + matv(ri,6)*vi(colind(ri,6)) &
          + matv(ri,7)*vi(colind(ri,7))
end do
```

→ The general variant runs at $\approx 91\%$ of the performance of the specialised version.

Vector friendly matrix storage formats

→ The constructor supplied math library NLC implements two sparse matrix formats: **Ellpack** and **MJAD**. Neither of them gets significantly closer to the roofline bound.

→ Despite the additional computations, the specialised variant is faster (in wall clock time) than the NLC Ellpack routines on our test cases.

Ellpack:

→ For vertex-based discretization schemes on unstructured meshes, plain vanilla Ellpack is **not** the most appropriate choice.

→ **But: Face-based schemes like CDO yield (almost) constant row length on unstructured, single-element type meshes.**

→ On multi-element meshes, one could consider a sliced Ellpack format, or the constructor supplied MJAD.

Single core performance

Wall clock ratio for one CG iteration on

→ Intel Xeon Gold 6140, Intel compilers 2019.0.045 and spmv from MKL 2019.0.045

→ NEC SX-Aurora T10B, NEC compilers, in-house Ellpack spmv

Mesh type	**discr.**	**# rows**	**# nnz**	$r = \frac{T(x86)}{T(Aurora)}$
hexahedra	FV	32768	223232	8,30
hexahedra	FV	2097152	14581760	16,63
tetrahedra	CDOfb	124988	857612	15,74
tetrahedra	CDOfb	309248	2139392	14,47
prisms	CDOfb	326400	2886400	13,98
non-conf. hexahedra	CDOfb	592896	13166976	12,20
hexahedra	CDOfb	798720	8663040	14,67

The Stream benchmark does not cover the indirect memory acces patterns of a SpMV, which explains why the difference in the Stream results (x30) is larger than that of the CG case ($\approx$ x15).

To sum up

Conclusion:

- → The Stream benchmark does not test irregular memory acces patterns.
  ⇒ Additional tests needed to predict performance on new architectures.
- → High bandwidth memory architectures are one way forward to reduce run times for iterative solvers on sparse matrices.
- → For straightforward access patterns, the results on the NEC SX-Aurora are optimal, as predicted by the roofline model.
- → In the case of indirect and non-contiguous memory access patterns, the effective memory bandwidth drops by a factor of 2.
- → Data structures have to be chosen carefully on the vector architecture, otherwise the performance penalty is significant (factor x30 between different sparse matrix formats).
- → The software environment (Fortran, C, C++, OpenMP, MPI) is legacy code (and developer!) friendly, but as with every optimizing compiler, certain constraints have to be taken into account.

Outlook:

- → Software evolution: MPI between vector card(s) and x86 host to be installed soon.
- → THE open question: Which preconditioners can benefit from such a HBM architecture?

Thank you for your attention!

Questions?