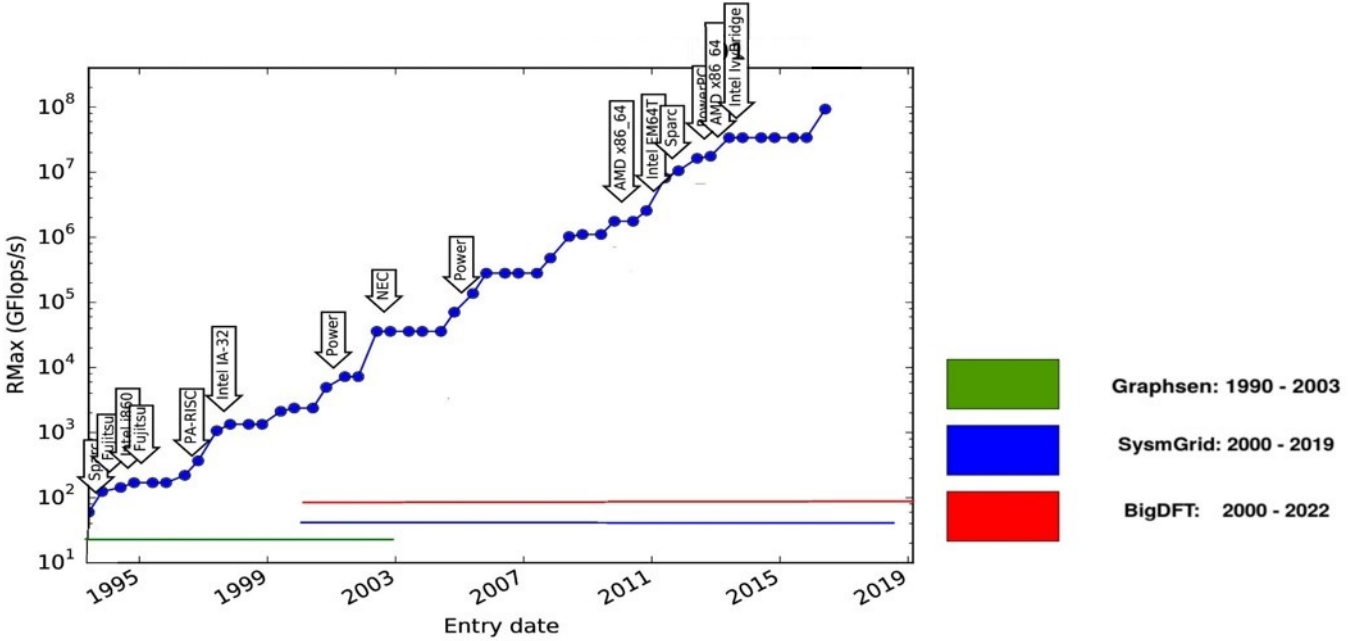
Data-Layout Optimization based on Memory-Access-Pattern Analysis for Source-Code Performance Improvement

Authors: Riyane SID LAKHDAR, Henri-Pierre CHARLES, Maha KOOLI
Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France

23rd International Workshop on Software and Compilers for Embedded Systems (SCOPES '20)

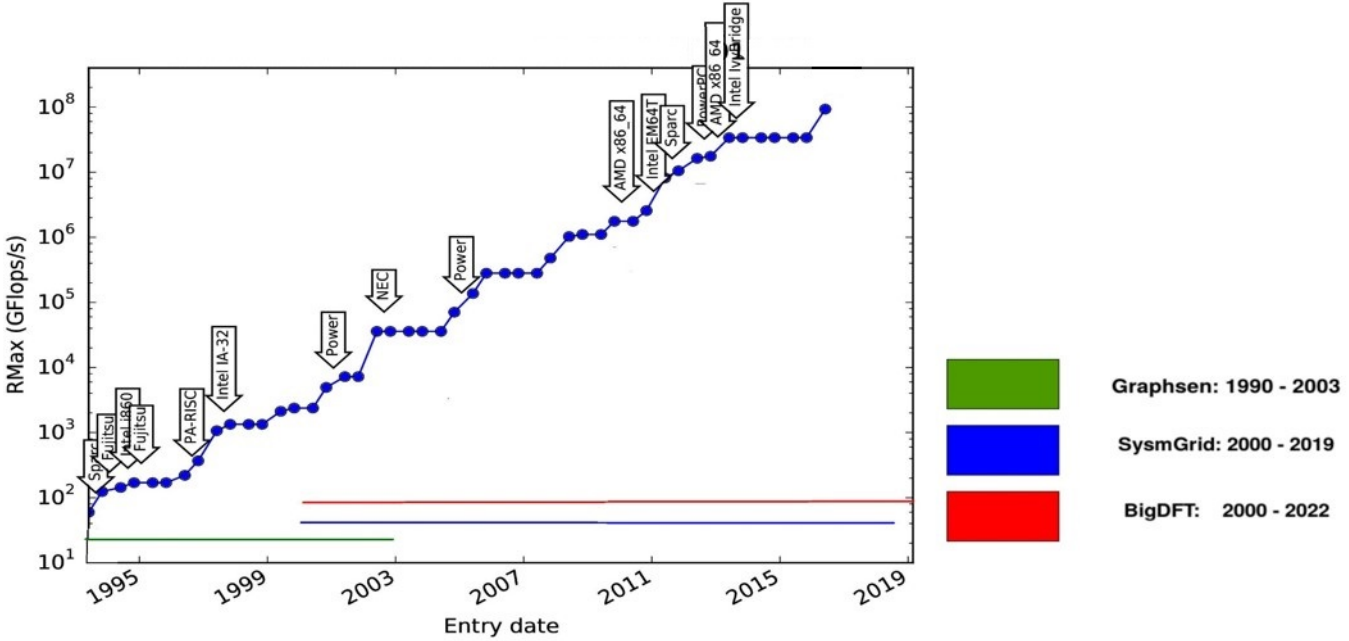
Sankt Goar, Germany | May 26th 2020

CONTEXT AND MOTIVATIONS



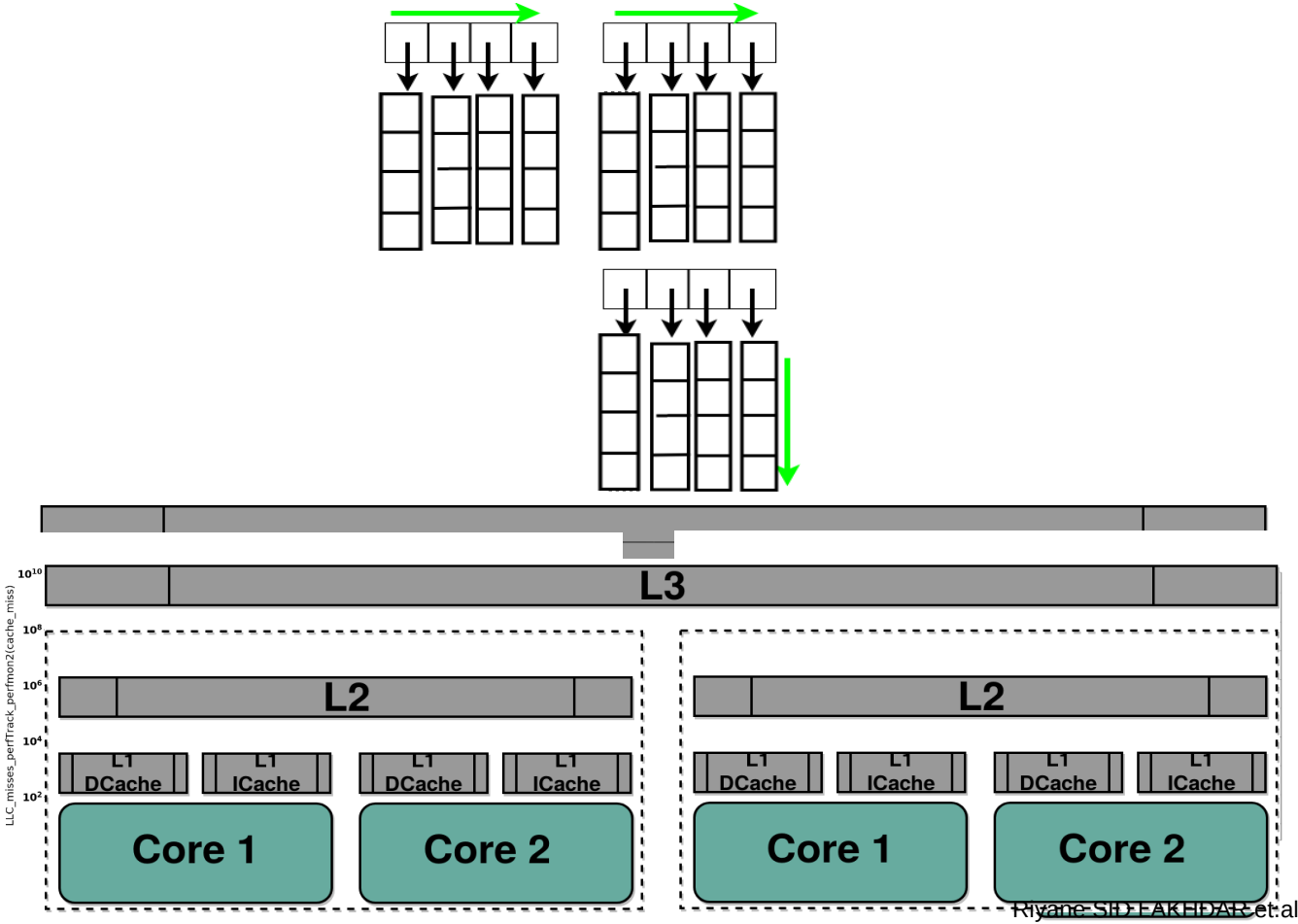
- **Scientific application crosses different HW technologies**

CONTEXT AND MOTIVATIONS

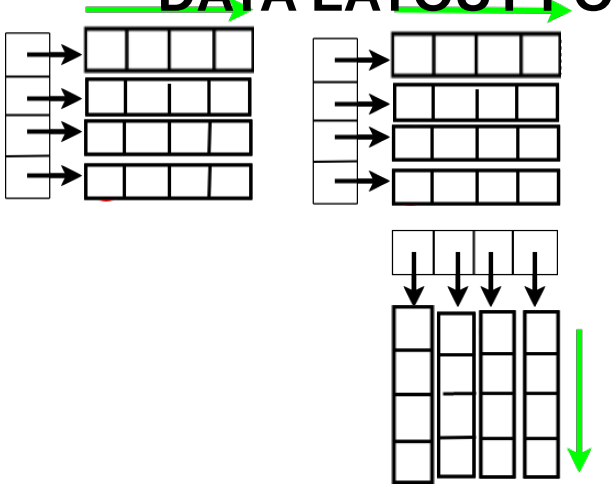


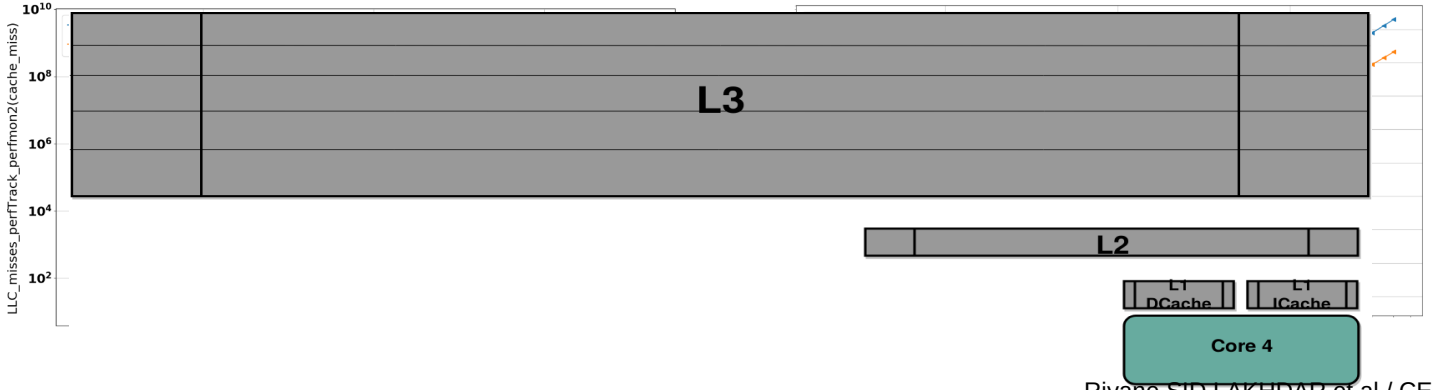
- **Scientific application crosses different HW technologies**
- **Important time/engineering effort to keep apps adapted to HW**

PROBLEM:
DATA LAYOUT FOR HW/SW PERFORMANCE



PROBLEM:
DATA LAYOUT FOR HW/SW PERFORMANCE





OBJECTIVE AND METHOD

Problem:

- possible implementations for the matrix data-layout
- Overall performances deeply impacted[SidLakhdar_2019]

[SidLakhdar_2019] Sid Lakhdar Riyane et al. “Toward Modeling of Cache-Miss Ratio for Dense-Data-Access-Based Optimization”. In RSP 2019. ACM.

OBJECTIVE AND METHOD

Problem:

- possible implementations for the matrix data-layout
- Overall performances deeply impacted[SidLakhdar_2019]

Objective:

Automatically detect the most efficient data-layout implementation:

- For each variable
- With regards to the host hardware (memory)

Method:

Map the detected memory-access pattern with a known optimized implementation

[SidLakhdar_2019] Sid Lakhdar Riyane et al. “Toward Modeling of Cache-Miss Ratio for Dense-Data-Access-Based Optimization”. In RSP 2019. ACM.

State of the art: Pattern detection, usage and DLD

HARDSI: Hardware Adapted Restructuring of Data Structure Implementation

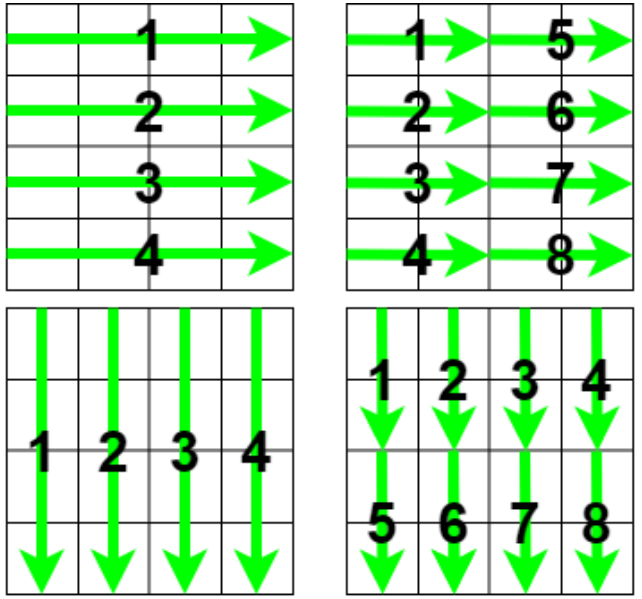
Experimental Results

Enhancing HARDSI with Data-cache modeling

STATE OF THE ART: PATTERN DETECTION

What is a memory access pattern:

“smallest set of consecutive accesses (read and write) to a given data structure that can be repeated in order to represent the total accesses to the data structure.” [xu_2019]



The accesses are either:

a) Addresses (virtual/physical)
b) Indexes (e.g. array)
c) Transformation of a) or b)

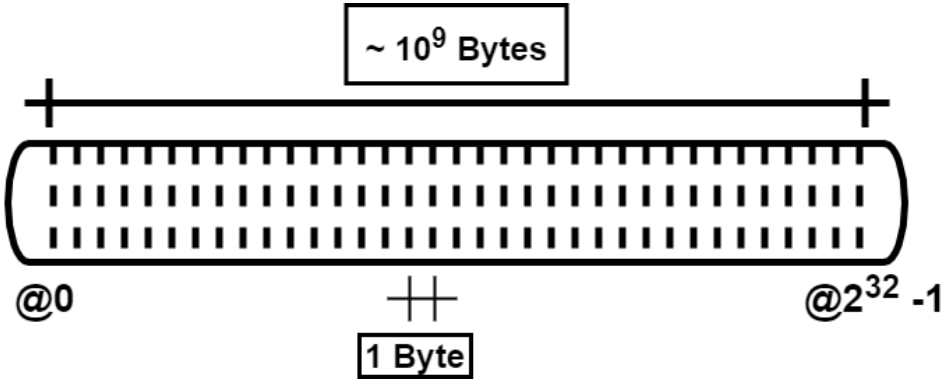
[xu_2019] Xu Zhixing, Ray Sayak, Subramanyan Pramod and Malik Sharad: «Malware detection using machine learning based analysis of virtual memory access patterns». In Proceedings of the Conference on Design, Automation & Test in Europe.

STATE OF THE ART: PATTERN DETECTION

Detection of memory-access pattern:

- Intensively used by memory pre-fetchers
- Used to predict the next addresses to be accessed [Wilkerson_19]. Exemple:

➢ Toddler[Nistor_13],
➢ QUAD[Ostadzadeh_15],
➢ Aristole[Fang_17]



[Wilkerson_2019] Christopher B Wilkerson et al. 2019. Instruction and logic for software hints to improve hardware prefetcher effectiveness. US Patent 10,229,060.
[Nistor_13] Nistor Adrian, et al. «Toddler: Detecting performance problems via similar memory-access patterns». In Proceedings of the ICSE'13, IEEE Press.
[Ostadzadeh_15] Ostadzadeh S Arash, et al. «Quad: a memory access pattern analyser». In ISARC.
[Fang_17] Fang Jianbin, et al. «Aristotle: A performance impact indicator for the OpenCL kernels using local memory». In the Scientific Programming journal.

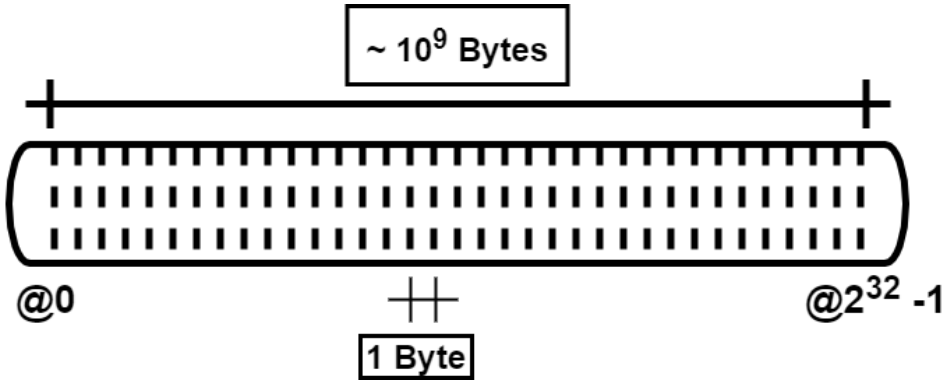
STATE OF THE ART: PATTERN DETECTION

Detection of memory-access pattern:

- Intensively used by memory pre-fetchers
- Used to predict the next addresses to be accessed [Wilkerson_19]. Exemple:
  - Toddler[Nistor_13],
  - QUAD[Ostadzadeh_15],
  - Aristole[Fang_17]

Problem:

- Granularity ~ Bytes
- Does not scale for a data structure

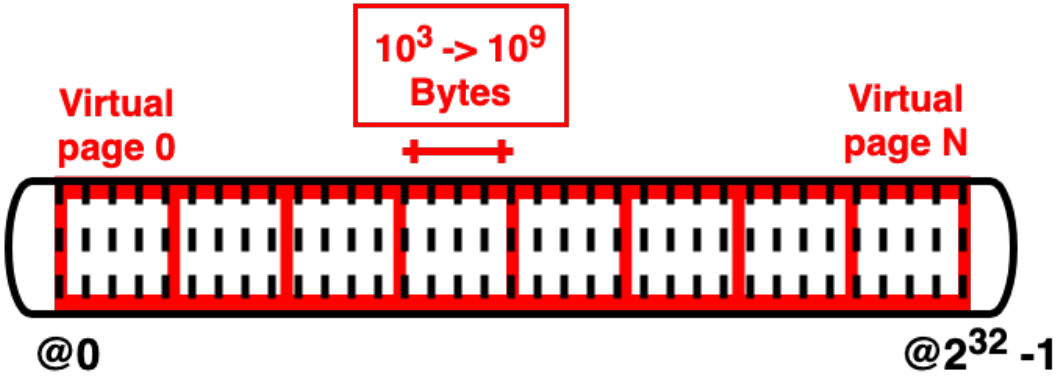


[Wilkerson_2019] Christopher B Wilkerson et al. 2019. Instruction and logic for software hints to improve hardware prefetcher effectiveness. US Patent 10,229,060.
[Nistor_13] Nistor Adrian, et al. «Toddler: Detecting performance problems via similar memory-access patterns». In Proceedings of the ICSE'13, IEEE Press.
[Ostadzadeh_15] Ostadzadeh S Arash, et al. «Quad: a memory access pattern analyser». In ISARC.
[Fang_17] Fang Jianbin, et al. «Aristotle: A performance impact indicator for the OpenCL kernels using local memory». In the Scientific Programming journal.

STATE OF THE ART: PATTERN DETECTION

Profiling of memory-access pattern:

- Mainly used in the detection of malware or fault injection
- Exemple: [Xu_2019]



[xu_2019] Xu Zhixing, Ray Sayak, Subramanyan Pramod and Malik Sharad: «Malware detection using machine learning based analysis of virtual memory access patterns». In Proceedings of the Conference on Design, Automation & Test in Europe.

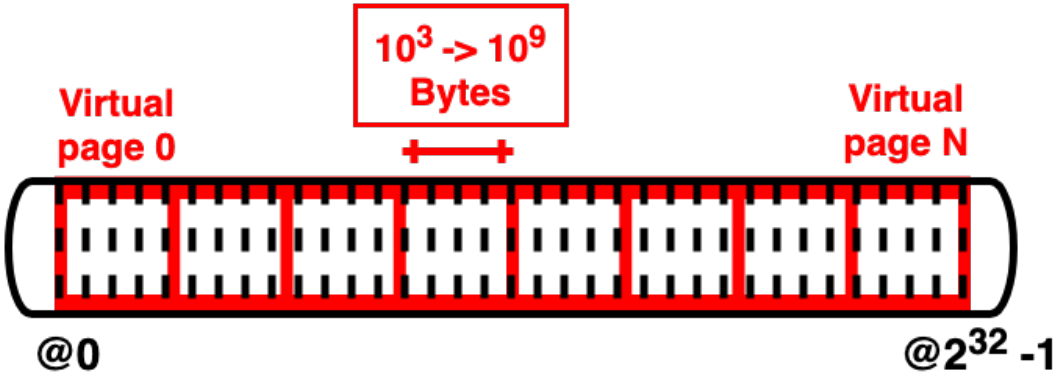
STATE OF THE ART: PATTERN DETECTION

Profiling of memory-access pattern:

- Mainly used in the detection of malware or fault injection
- Exemple: [Xu_2019]

Problem:

- Granularity: virtual pages
- Does not scale for a data structure



[xu_2019] Xu Zhixing, Ray Sayak, Subramanyan Pramod and Malik Sharad: «Malware detection using machine learning based analysis of virtual memory access patterns». In Proceedings of the Conference on Design, Automation & Test in Europe.

STATE OF THE ART: DATA-LAYOUT DECISION PROBLEM

	Granularity			Optimization time		Target memory/application	
	Scalar variable	Allocator block	Virtual page	compile time	run time	Portable to new memories	Portable to new applications
[Lian_05]	(*)			(*)			
[Shoushtari_18]		(*)		(*)			
[Serrano_19]			(*)	(*)			
[Doosan_08]	(*)				(*)		
[Kandemir_01]			(*)		(*)		
[Cooper_98]			(*)	(*)			
[Issenin_06]		(*)			(*)		

[15] Lian Li et al. 2005. Memory coloring: A compiler approach for scratchpad memory. In PACT.
[18] Abdolmajid Namaki Shoushtari. 2018. Software Assists to On-chip Memory Hierarchy of Manycore Embedded Systems. Ph.D. Dissertation. UC Irvine.
[22] Manuel Serrano et al. 2019. Property caches revisited. In CC.
[2] Doosan Cho et al. 2008. Compiler driven data layout optimization for regular/irregular array access patterns. ACM.
[9] Ilya Issenin et al. 2006. Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies. In DAC.
[10] Mahmut Kandemir et al. 2001. Dynamic management of scratch-pad memory space. In DAC. IEEE.
[3] Keith D Cooper and Timothy J Harvey. 1998. Compiler-controlled memory. In SIGOPS OSR. ACM.

STATE OF THE ART: DATA-LAYOUT DECISION PROBLEM

	Granularity			Optimization time		Target memory/application	
	Scalar variable	Allocator block	Virtual page	compile time	run time	Portable to new memories	Portable to new applications
[Lian_05]	(*)			(*)			
[Shoushtari_18]		(*)		(*)			
[Serrano_19]			(*)	(*)			
[Doosan_08]	(*)				(*)		
[Kandemir_01]			(*)		(*)		
[Cooper_98]			(*)	(*)			
[Issenin_06]		(*)			(*)		

Limitation:

- Require human intervention
- No direct code specialization to hardware

OUTLINES

State of the art: Pattern detection, usage and DLD

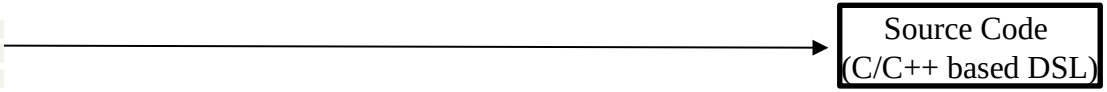
HARDSI: Hardware Adapted Restructuring of Data Structure Implementation

Experimental Results

Enhancing HARDSI with Data-cache modeling

SCIENTIFIC APPROACH

```
void matrixMult()
{
  MATRIX_DEFINE(int, a);
  MATRIX_DEFINE(int, b);
  MATRIX_DEFINE(int, res);


  int i,j,k, a0, b0;

  // Initialize the data structures
  MATRIX_ALLOCATE(int, N0, N1, a);

  ...
  for (j=0; j<N1; j++)
    for (i=0; i<N2; i++)
      for (k=0; k<N0; k++)
      {
        a0=MATRIX_GET(a,k,j);
        b0=MATRIX_GET(b,i,k);
        MATRIX_ADD(res,i,j,a0*b0);
      }
   ...
  // Free the data structures
  MATRIX_FREE(a,N0,N1, int);
```

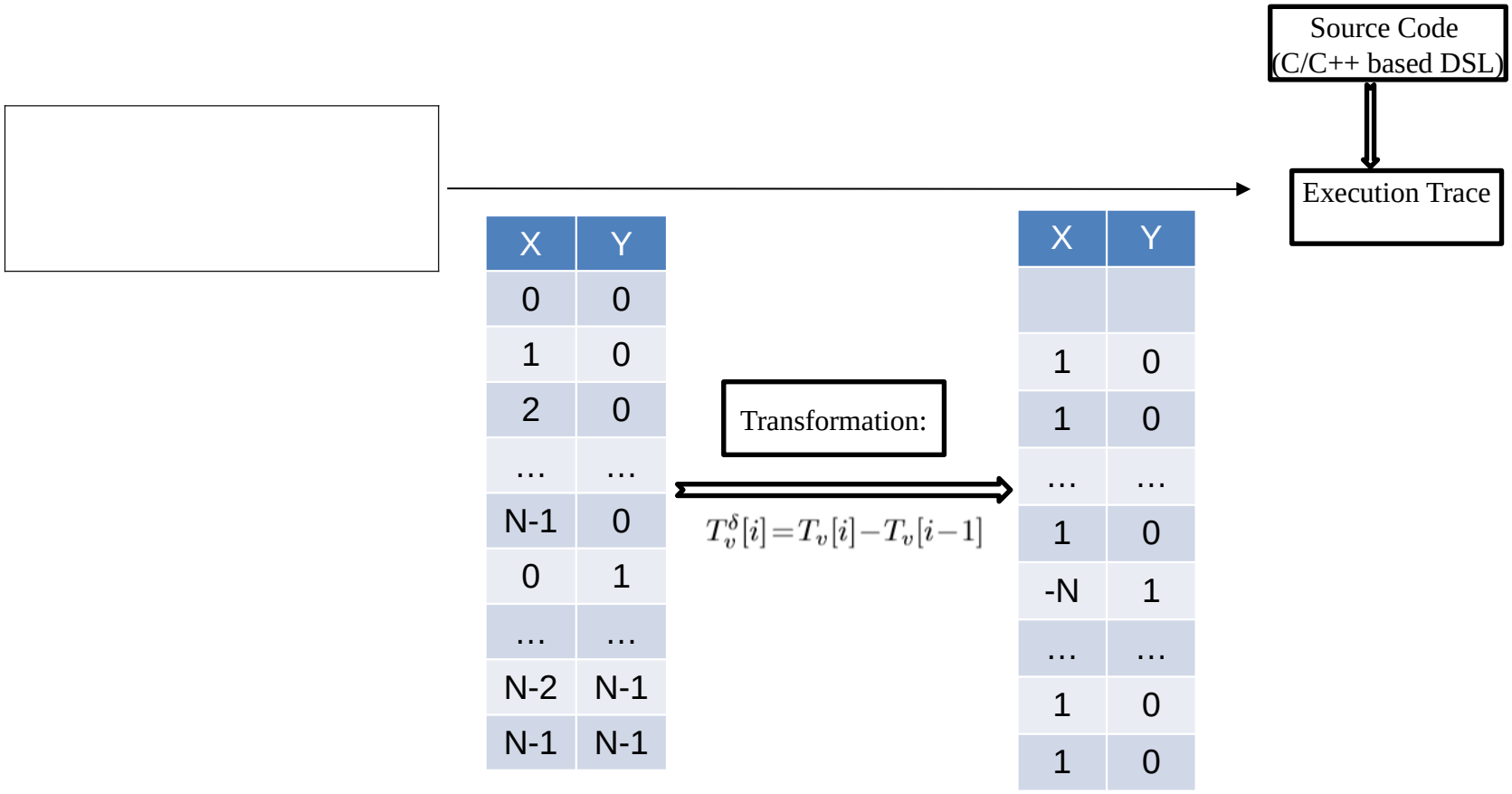


SCIENTIFIC APPROACH

SCIENTIFIC APPROACH

X	Y
0	0
1	0
2	0
…	…
N-1	0
0	1
…	…
N-2	N-1
N-1	N-1

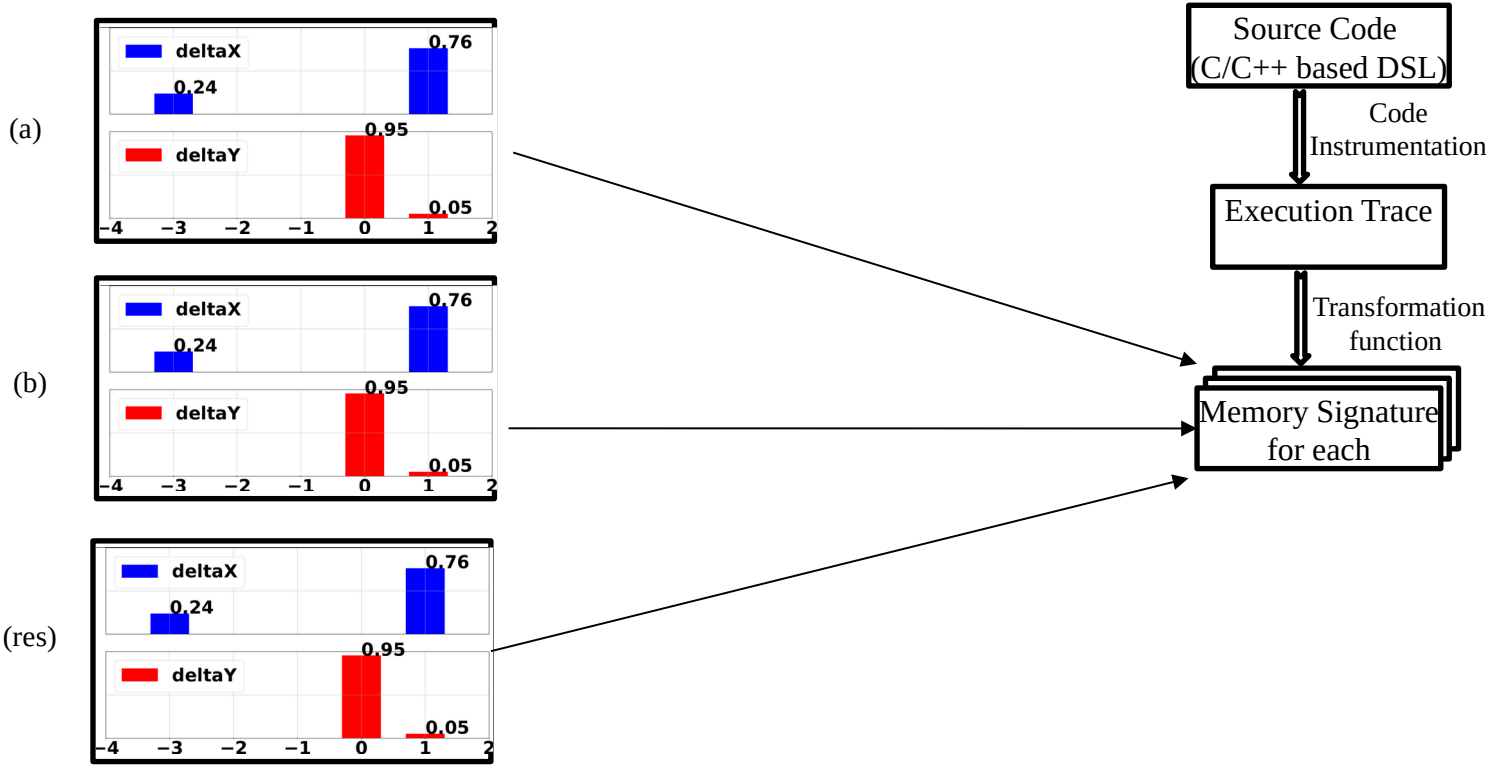
SCIENTIFIC APPROACH



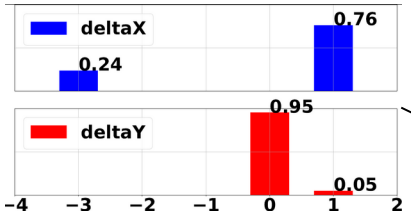
X	Y
0	0
1	0
2	0
…	…
N-1	0
0	1
…	…
N-2	N-1
N-1	N-1

X	Y
1	0
1	0
…	…
1	0
-N	1
…	…
1	0
1	0

SCIENTIFIC APPROACH



SCIENTIFIC APPROACH



Source Code (C/C++ based DSL)

Code Instrumentation

Execution Trace

Transformation function

Memory Signature for each

Correlation:

$$\rho(S,S') = \frac{cov(S,S')}{\sigma_S \sigma_{S'}} = \frac{\frac{1}{N}\sum_{i=0}^{N-1}(S[i]-\mathbb{E}_S)(S'[i]-\mathbb{E}_{S'})}{\sigma_S \sigma_{S'}}$$

Optimal Implementation of each

Inject optimal implementation of each variable

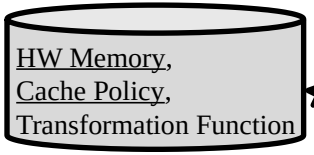
HW Memory,
Cache Policy,
Transformation Function

Data Base of known access-pattern signatures

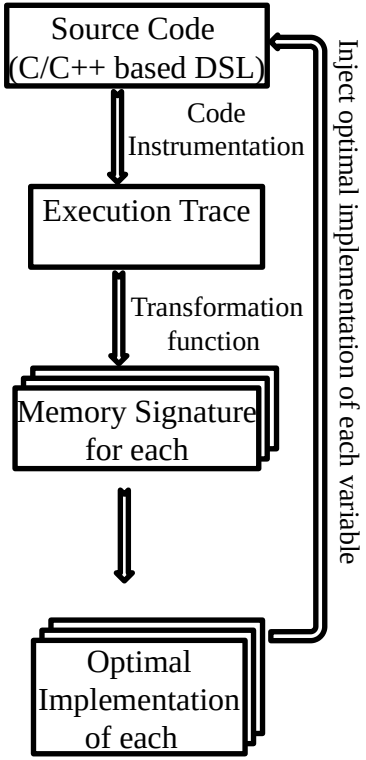
SCIENTIFIC APPROACH

```
MATRIX_DEFINE(int, a);
MATRIX_DEFINE(int, b);
MATRIX_DEFINE(int, res);


int i,j,k, a0, b0;

// Initialize the data structures
MATRIX_ALLOCATE(int, N0, N1, a);
...
for (j=0; j<N1; j++)
  for (i=0; i<N2; i++)
    for (k=0; k<N0; k++)
    {
      a0=MATRIX_GET(a,k,j);
      b0=MATRIX_GET(b,i,k);
      MATRIX_ADD(res,i,j,a0*b0);
    }
...
// Free the data structures
MATRIX_FREE(a,N0,N1, int);
```

```
int **a;
int **b;
int **res;
int s =sizeof(int);
int sp=sizeof(int*);
int i,j,k, a0, b0;

// Initialize the data structures
a = (int**)malloc(N1*sp);
for (i=0; i<N1; i++)
  a[i]=(int*)malloc(N0*s);
...
for (j=0; j<N1; j++)
  for (i=0; i<N2; i++)
    for (k=0; k<N0; k++)
    {
      a0=a[j][k];
      b0=b[i][k];
      res[j][i]+=a0*b0;
    }
...
// Free the data structures
for (i=0; i<N0; i++)
  free(a[i]);
free(a);
```



Data Base of known access-pattern signatures



OUTLINES

State of the art: Pattern detection, usage and DLD

HARDSI: Hardware Adapted Restructuring of Data Structure Implementation

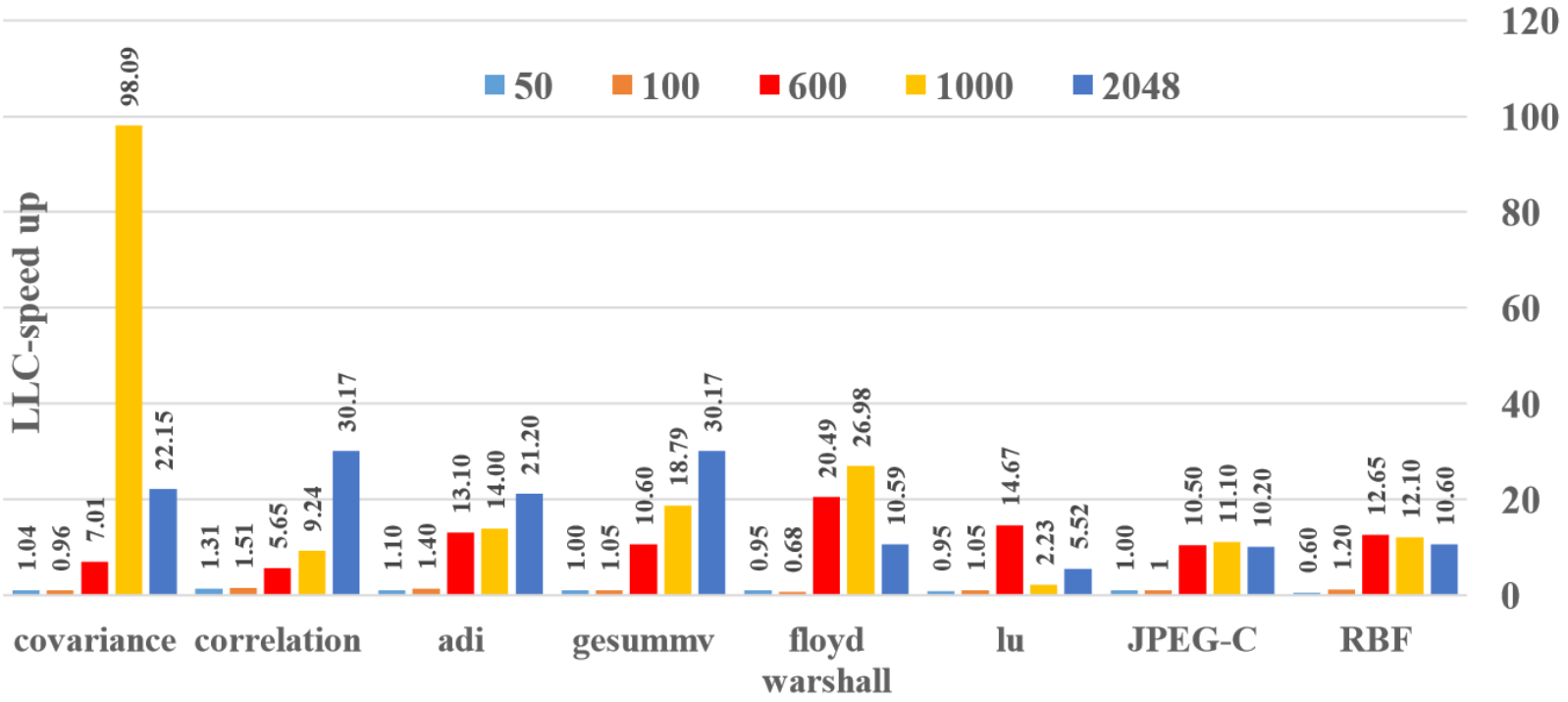
Experimental Results

Enhancing HARDSI with Data-cache modeling

Impact of the optimized implementation

	Line-maj	Column-maj	Stencil	Line-maj block-line	Line-maj block-diag
covariance	✓				
correlation	✓				
adi	✓		✓		
gesummv	✓	✓			
floyd-warshall		✓			
lu	✓	✓	✓		
JPEG-C				✓	✓
RBF	✓	✓		✓	

Impact of the optimized implementation: LLC



Interest of HARDSI: finding the best known implementation

Matrix size	50	100	600	1000	2048	50	100	600	1000	2048	50	100	600	1000	2048	50	100	600	1000	2048
	covariance					correlation					adi					gesummv				
Speed up HARDSI	1.0	1.1	3.2	8.3	12.2	1.0	1.0	3.1	7.7	12.1	1.4	1.6	1.6	1.4	2.3	1.4	1.3	7.1	12.2	12.5
Speed up Best SoA	=	=	=	=	=	=	=	=	=	=	1.0	1.0	1.0	1.0	1.3	=	=	=	=	=
Reference Best SoA	[1]					[10]					[10]					[1]				
	floyd-warshall					lu					JPEG-C					RBF				
Speed up HARDSI	3.9	5.4	28.9	48.9	29.4	1.0	1.0	1.5	4.0	6.3	1.0	3.0	7.9	10.4	12	2.1	10.9	34.5	42.5	45.9
Speed up Best SoA	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=
Reference Best SoA	[10]					[1]					[14]					[27]				

[1] Mohamed Benabderrahmane et al. 2010. The polyhedral model is more widely applicable than you think. In CC.
[10] Mahmut Kandemir et al. 2001. Dynamic management of scratch-pad memory space. In DAC. IEEE.
[14] Alain M Leger et al. 1991. JPEG still picture compression algorithm. Optical Engineering (1991).
[27] Qingxiong Yang. 2012. Recursive bilateral filtering. In ECCV.

Summary:

- A novel method to solve the data-layout decision problem
- Maps accurately a memory-access pattern with the corresponding optimized implementation

Perspectives:

- Support new hardware memories (e.g. scratchpad, NVM, In-memory computing)
- Reduce the complexity to support more complexe code (ML, AI) at compiler-scale time

Centre de Saclay
Nano-Innov PC 172 - 91191 Gif sur Yvette Cedex

Find the ideal implementation of each variable's data-structure

```
void matrixMult(Matrix a, Matrix b, Matrix res)
{
     for (int y=0; y<N1; y++)
     {
          for (int x=0; x<N0; x++)
          {
               MATRIX_SET(res, x, y, 0);
               for (int k=0; k<N0; k++)
               {
                    int tmpA = MATRIX_GET(a, k, y);
                    int tmpB = MATRIX_GET(b, x, k);
                    MATRIX_ADD(res, x, y, tmpA*tmpB);
               }
          }
     }
}
```

```
/**
 * Matrix defined per lines:
 * Array where each cell is a pointer to a line of the matrix
 */
   #define LINE_MAJ_GET(m, x, y)              m[y][x]
   #define LINE_MAJ_SET(m, x, y, val)         m[y][x] = val
   #define LINE_MAJ_ADD(m, x, y, val)         m[y][x] += val

/**
 * Matrix defined per columns:
 * Array where each cell is a pointer to a column of the matrix
 */
   #define COLUMN_MAJ_GET(m, x, y)            m[x][y]
   #define COLUMN_MAJ_SET(m, x, y, val)       m[x][y] = val
   #define COLUMN_MAJ_ADD(m, x, y, val)       m[x][y] += val

/**
 * Matrix defined per columns:
 * Array where each cell is a pointer to a column of the matrix
 */
   #define DIAG_X(x, y)                       (x == y) ? x : (x > y) ? y         : x
   #define DIAG_Y(x, y)                       (x == y) ? 0 : (x > y) ? 1+x+x-y-y : y+y-x-x
   #define DIAG_MAJ_GET(m, x, y)              m[DIAG_X(x, y)][DIAG_Y(x, y)]
   #define DIAG_MAJ_SET(m, x, y, val)         m[DIAG_X(x, y)][DIAG_Y(x, y)]  = val;
   #define DIAG_MAJ_ADD(m, x, y, val)         m[DIAG_X(x, y)][DIAG_Y(x, y)] += val
```

OUTLINES

State of the art: Pattern detection, usage and DLD

HARDSI: Data-Structure Implementation selection

Experimental Results

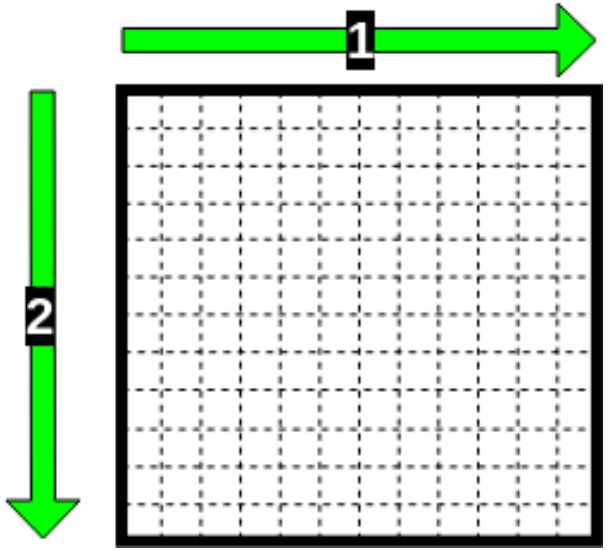
Perspectives: Data-cache modeling

Use case example: JPEG compression

```
void jpegCompression(rgbColor **imageMatrix)
{
        rgb_to_YVbCr    (imageMatrix);
        preDCT          (imageMatrix);
        DCT             (imageMatrix);
        quantization    (imageMatrix);
        encoding        (imageMatrix);

}
```

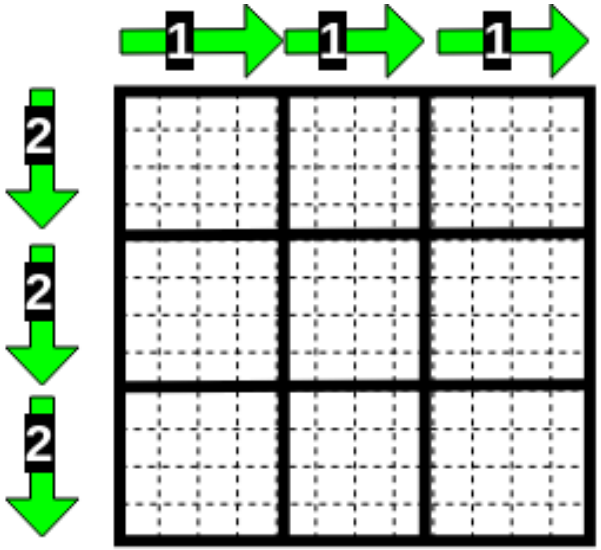
Use case example: JPEG compression

```
void jpegCompression(rgbColor **imageMatrix)
{
        rgb_to_YVbCr    (imageMatrix);
        preDCT          (imageMatrix);
        DCT             (imageMatrix);
        quantization    (imageMatrix);
        encoding        (imageMatrix);

}
```



Use case example: JPEG compression

```
void jpegCompression(rgbColor **imageMatrix)
{
        rgb_to_YVbCr    (imageMatrix);
        preDCT          (imageMatrix);
        DCT             (imageMatrix);
        quantization    (imageMatrix);
        encoding        (imageMatrix);

}
```



Use case example: JPEG compression

```
void jpegCompression(rgbColor **imageMatrix)
{
        rgb_to_YVbCr    (imageMatrix);
        preDCT          (imageMatrix);
        DCT             (imageMatrix);
        quantization    (imageMatrix);
        encoding        (imageMatrix);

}
```