Lecture 9: Dense Matrices and Decomposition

CSCE 569 Parallel Computing

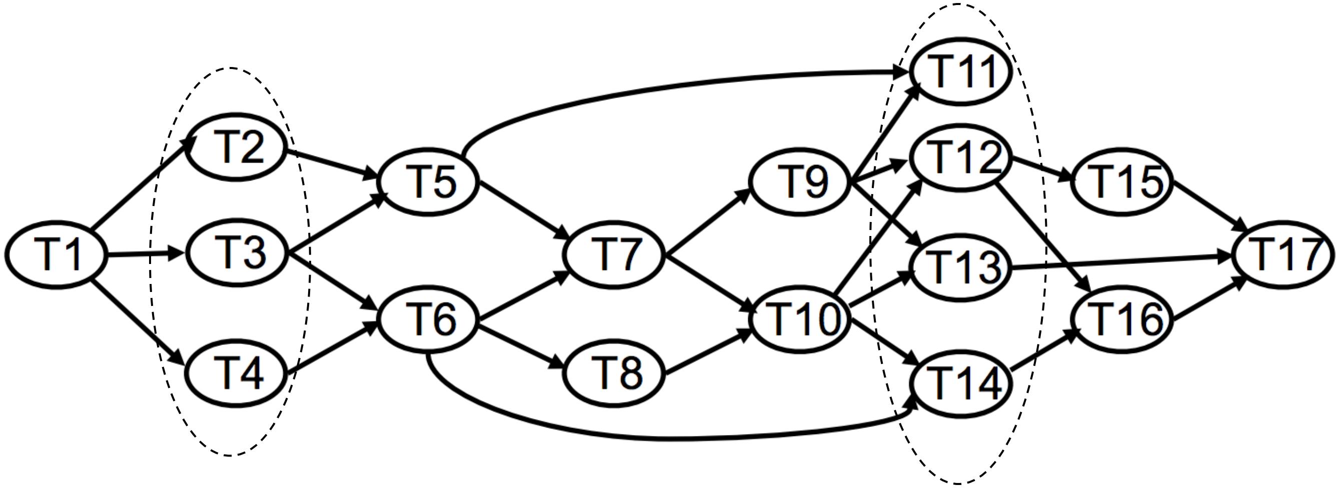
Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
http://cse.sc.edu/~yanyh

Review: Parallel Algorithm Design and Decomposition

- **Introduction to Parallel Algorithms**
  - Tasks and Decomposition
  - Processes and Mapping
- Decomposition Techniques
  - Recursive Decomposition
  - Data Decomposition
  - Exploratory Decomposition
  - Hybrid Decomposition
- Characteristics of Tasks and Interactions
  - Task Generation, Granularity, and Context
  - Characteristics of Task Interactions.

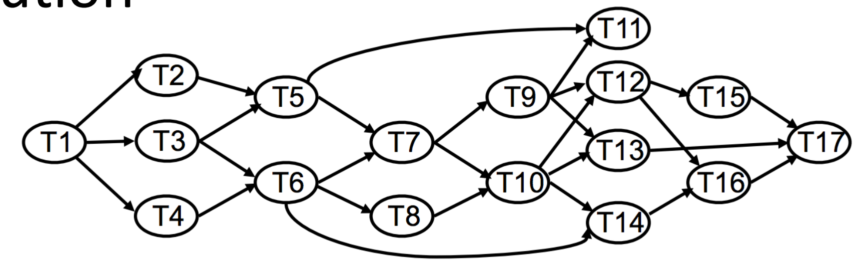
Decomposition, Tasks, and Dependency Graphs

- Decompose work into tasks that can be executed concurrently
- Decomposition could be in many different ways.
- Tasks may be of same, different, or even indeterminate sizes.
- Task dependency graph:
  - node = task
  - edge = control dependence, output-input dependency
  - No dependency == parallelism



Degree of Concurrency

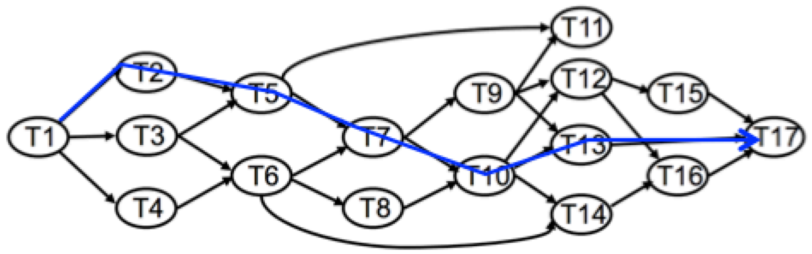
- Definition: the number of tasks that can be executed in parallel
- May change over program execution



- Metrics
  - *Maximum degree of concurrency*
    - Maximum number of concurrent tasks at any point during execution.
  - *Average degree of concurrency*
    - The average number of tasks that can be processed in parallel over the execution of the program
    - **Speedup: serial_execution_time/parallel_execution_time**
- Inverse relationship of degree of concurrency and task granularity
  - Task granularity ↑(less tasks), degree of concurrency ↓
  - Task granularity ↓(more tasks), degree of concurrency ↑

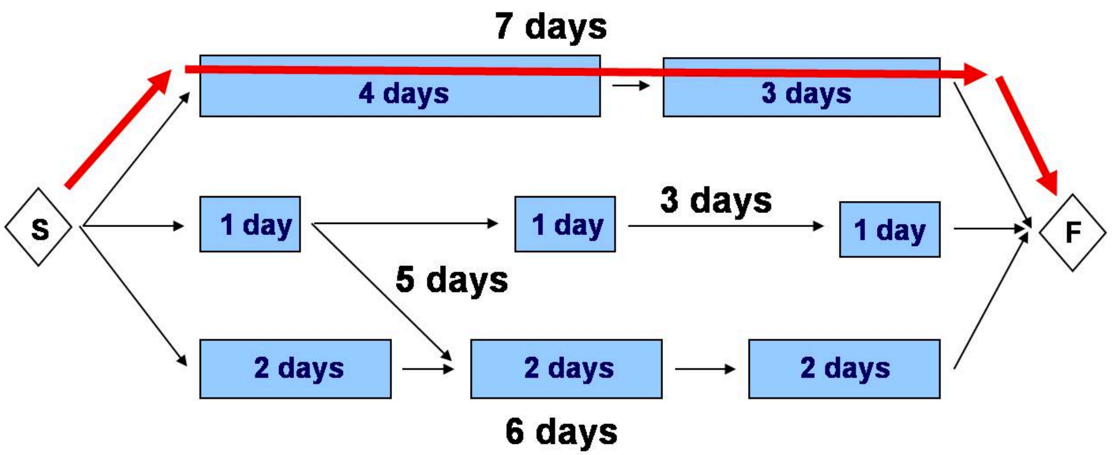
Critical Path Length

- **A directed path:** a sequence of tasks that must be serialized
  – Executed one after another



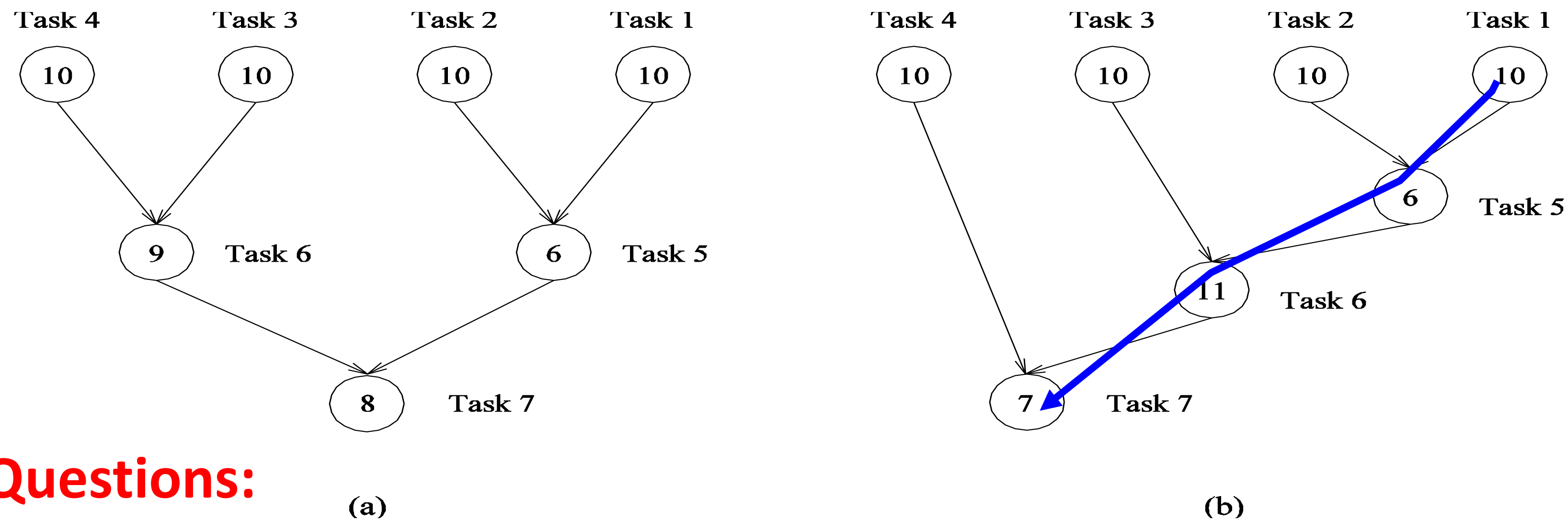
- Critical path:
  – The longest weighted path throughout the graph
- Critical path length: shortest time in which the program can be finished
  – Lower bound on parallel execution time

A building project



Critical Path Length and Degree of Concurrency

Database query task dependency graph



Questions:

What are the tasks on the critical path for each dependency graph?

What is the shortest parallel execution time?

How many processors are needed to achieve the minimum time?

What is the maximum degree of concurrency?

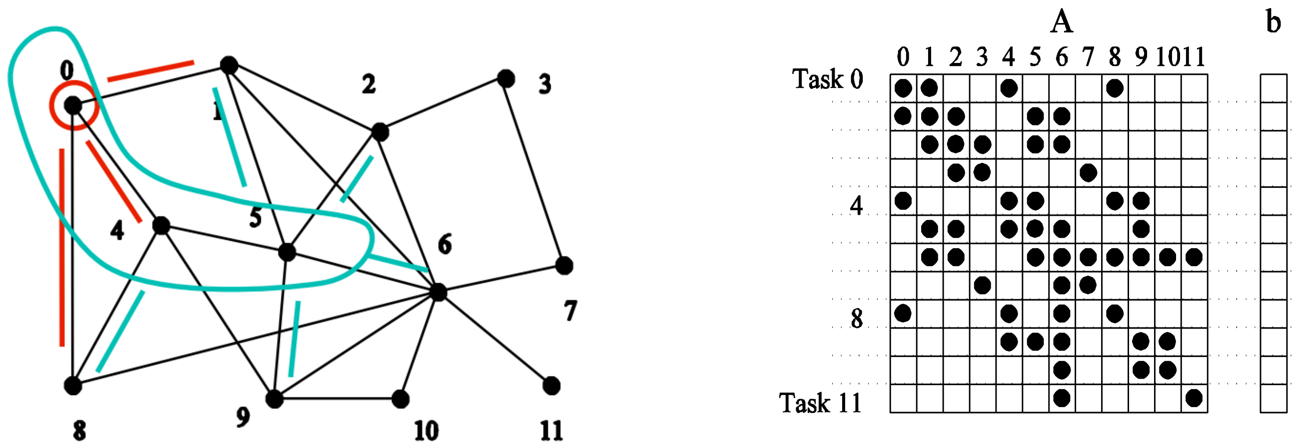
What is the average parallelism (average degree of concurrency)?

Total amount of work/(critical path length)

2.33 (63/27) and 1.88 (64/34)

Task Interaction Graphs, Granularity, and Communication

- Finer task granularity → more overhead of task interactions
  – Overhead as a ratio of useful work of a task
- Example: sparse matrix-vector product interaction graph



(a)

- Assumptions:
  – each dot (A[i][j]*b[j]) takes unit time to process
  – each communication (edge) causes an overhead of a unit time
- If node 0 is a task: communication = 3; computation = 4
- If nodes 0, 4, and 5 are a task: communication = 5; computation = 15
  – **coarser-grain decomposition → smaller communication/computation ratio (3/4 vs 5/15)**

Processes and Mapping

A good mapping must minimize parallel execution time by:

- Mapping independent tasks to different processes
  - Maximize concurrency
- Tasks on critical path have high priority of being assigned to processes
- Minimizing interaction between processes
  - mapping tasks with dense interactions to the same process.

- Difficulty: these criteria often conflict with each other
  - E.g. No decomposition, i.e. one task, minimizes interaction but no speedup at all!

Recursive Decomposition: Min

Finding the minimum in a vector using divide-and-conquer

```
procedure SERIAL_MIN (A, n)
    min = A[0];
    for i := 1 to n − 1 do
        if (A[i] < min) min := A[i];
    return min;
```

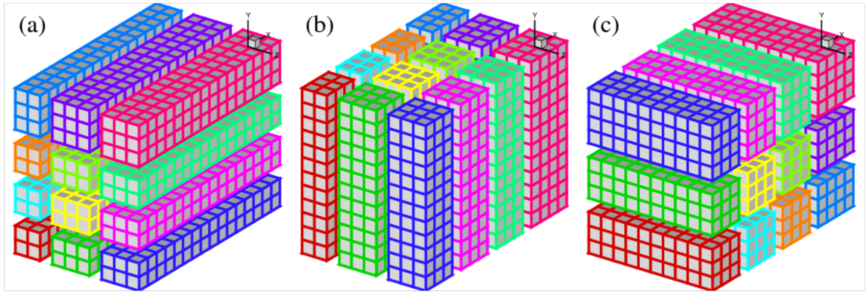
```
procedure RECURSIVE_MIN (A, n)
    if ( n = 1 ) then min := A [0]  ;
    else
        lmin := RECURSIVE_MIN (A, n/2 );
        rmin := RECURSIVE_MIN (&(A[n/2]), n - n/2);
        if (lmin  < rmin) then min := lmin;
        else min := rmin;
    return min;
```

Applicable to other associative operations, e.g. sum, AND …

Known as reduction operation

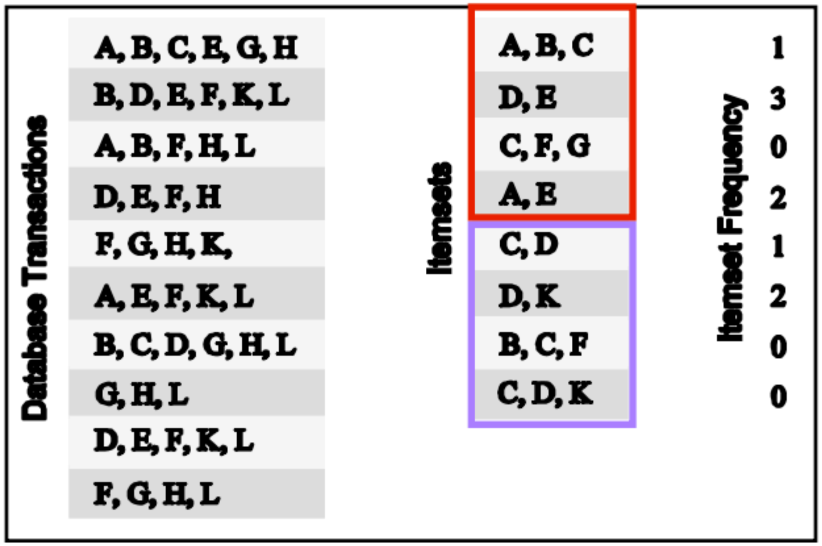
Data Decomposition
-- The most commonly used approach

- Steps:
  1. Identify the data on which computations are performed.
  2. Partition this data across various tasks.
     - Partitioning induces a decomposition of the problem, i.e. computation is partitioned
- Data can be partitioned in various ways
  – Critical for parallel performance
- Decomposition based on
  – output data
  – input data
  – input + output data
  – intermediate data

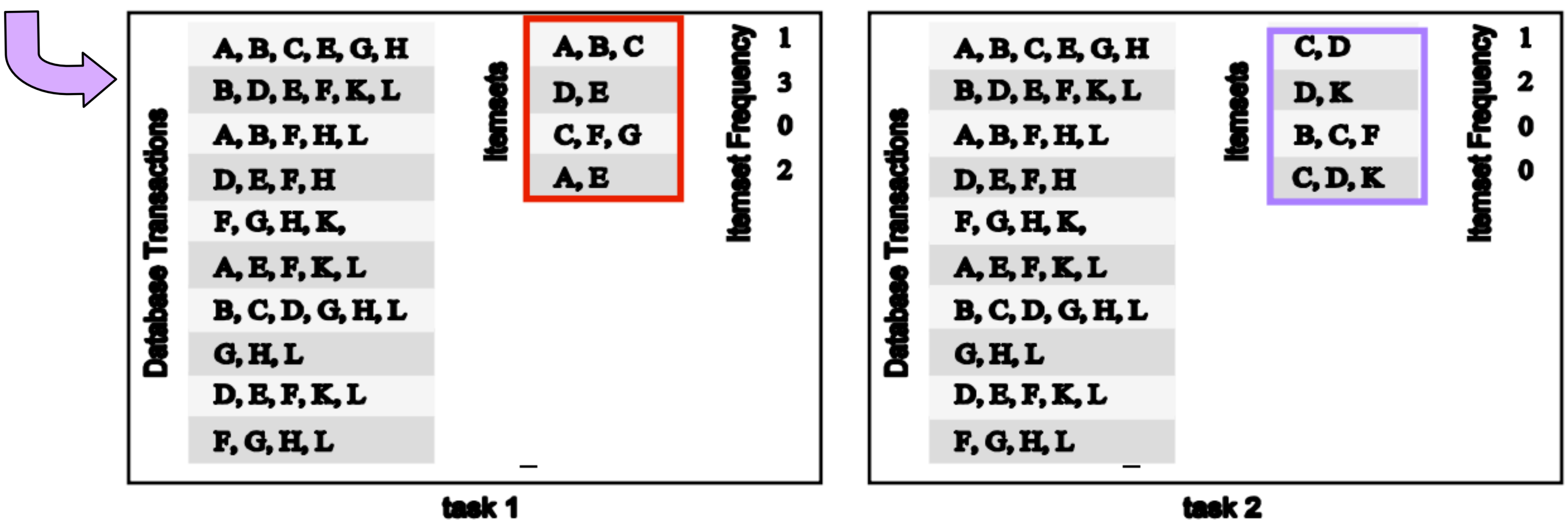


Output Data Decomposition: Example

Count the frequency of item sets in database transactions



- Decompose the item sets to count
  - each task computes total count for each of its item sets
  - append total counts for item sets to produce total count result



Input Data Partitioning: Example

Count the frequency of item sets in database transactions

Database Transactions
A, B, C, E, G, H
B, D, E, F, K, L
A, B, F, H, L
D, E, F, H
F, G, H, K,
A, E, F, K, L
B, C, D, G, H, L
G, H, L
D, E, F, K, L
F, G, H, L

Itemsets	Itemset Frequency
A, B, C	1
D, E	3
C, F, G	0
A, E	2
C, D	1
D, K	2
B, C, F	0
C, D, K	0

- **Partition computation by partitioning the set of transactions**

  —a task computes a local count for each item set for its transactions

task 1

Database Transactions
A, B, C, E, G, H
B, D, E, F, K, L
A, B, F, H, L
D, E, F, H
F, G, H, K,

Itemsets	Itemset Frequency
A, B, C	1
D, E	2
C, F, G	0
A, E	1
C, D	0
D, K	1
B, C, F	0
C, D, K	0

task 2

Database Transactions
A, E, F, K, L
B, C, D, G, H, L
G, H, L
D, E, F, K, L
F, G, H, L

Itemsets	Itemset Frequency
A, B, C	0
D, E	1
C, F, G	0
A, E	1
C, D	1
D, K	1
B, C, F	0
C, D, K	0

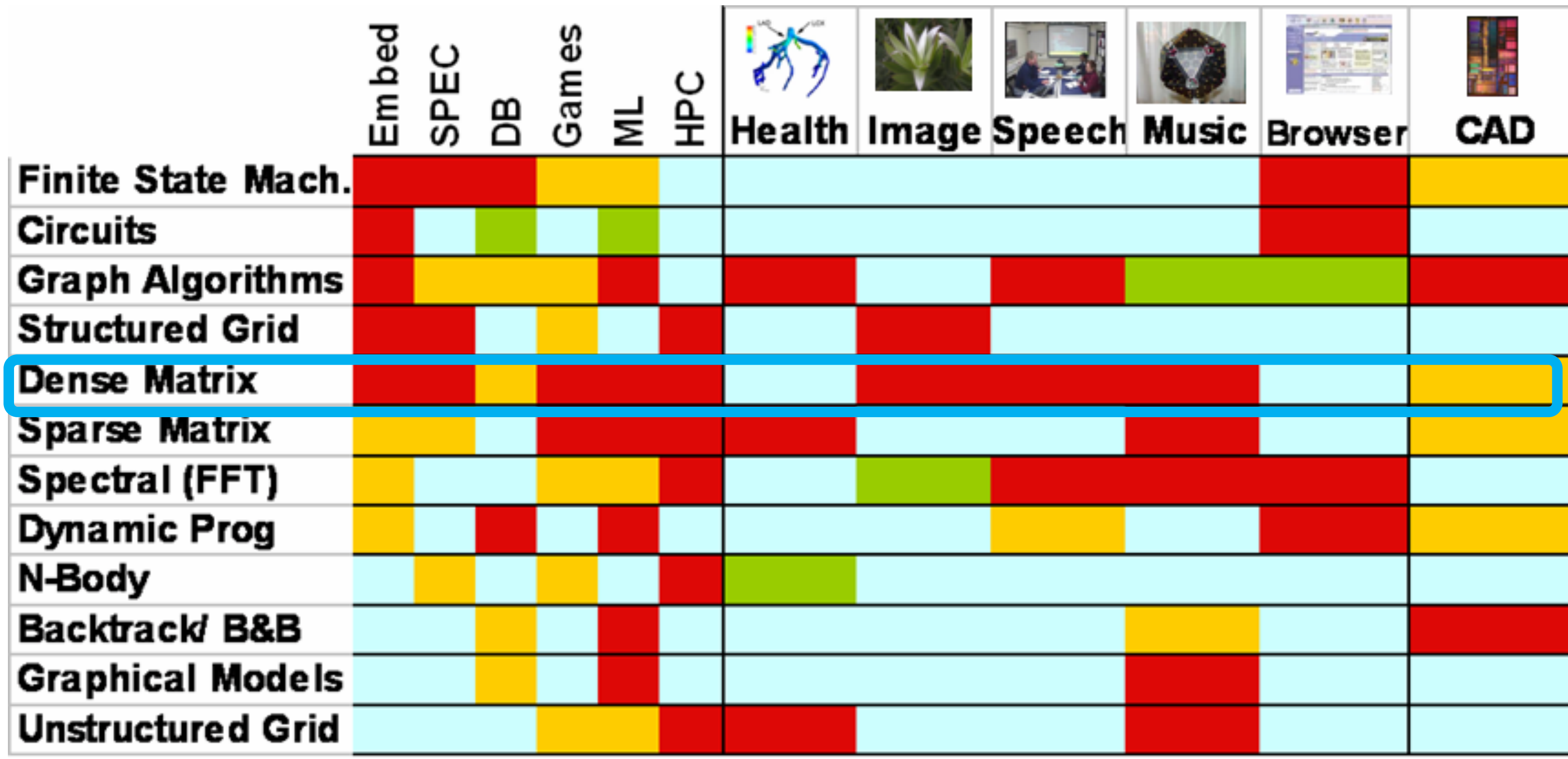
—sum local count vectors for item sets to produce total count vector

Dense matrix algorithms

- **Dense linear algebra and BLAS**
- Image processing/stencil
- Iterative methods

Motifs

The Motifs (formerly "Dwarfs") from "The Berkeley View" (Asanovic et al.) form key computational patterns



The Landscape of Parallel Computing Research: A View from Berkeley
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

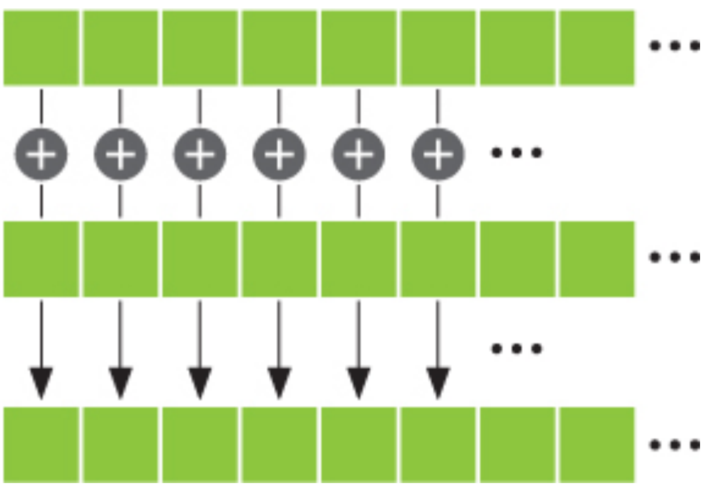
Dense linear algebra

- Software library solving linear system

- BLAS (Basic Linear Algebra Subprogram)
  – Vector, matrix vector, matrix matrix
- Linear Systems: $Ax=b$
- Least Squares: choose x to minimize $||Ax-b||_2$
  – Overdetermined or underdetermined
  – Unconstrained, constrained, weighted
- Eigenvalues and vectors of Symmetric Matrices
  - Standard ($Ax = \lambda x$), Generalized ($Ax=\lambda Bx$)
- Eigenvalues and vectors of Unsymmetric matrices
  - Eigenvalues, Schur form, eigenvectors, invariant subspaces
  - Standard, Generalized
- Singular Values and vectors (SVD)
  – Standard, Generalized
- Different matrix structures
  – Real, complex; Symmetric, Hermitian, positive definite; dense, triangular, banded …
- Level of detail
  – Simple Driver
  – Expert Drivers with error bounds, extra-precision, other options
  – Lower level routines ("apply certain kind of orthogonal transformation", matmul…)

BLAS (Basic Linear Algebra Subprogram)

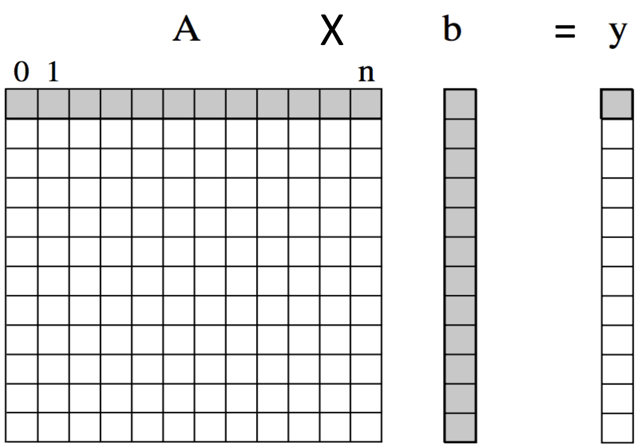
- BLAS 1, 1973-1977
  - 15 operations (mostly) on vectors (1-d array)
    - "AXPY" (y = α·x + y), dot product, scale (x = α·x)
  - Up to 4 versions of each (S/D/C/Z), 46 routines, 3300 LOC
  - **Why BLAS 1 ? They do $O(n^1)$ ops on $O(n^1)$ data:** AXPY
    - 2n flops on 3n read/writes
    - Computational intensity = (2n)/(3n) = 2/3



```
void axpy_base(int N, REAL Y[], REAL X[], REAL a) {
     int i;
     for (i = 0; i < N; ++i)
         Y[i] += a * X[i];
}
```

BLAS 2

- BLAS 2, 1984-1986
  - 25 operations (mostly) on matrix/vector pairs
  - "GEMV": $y = \alpha \cdot A \cdot x + \beta \cdot x$, "GER": $A = A + \alpha \cdot x \cdot yT$, $x = T\text{-}1 \cdot x$
  - Up to 4 versions of each (S/D/C/Z), 66 routines, 18K LOC
- Why BLAS 2 ? They do $O(n^2)$ ops on $O(n^2)$ data
  - Computational intensity still just $\sim(2n^2)/(n^2) = 2$



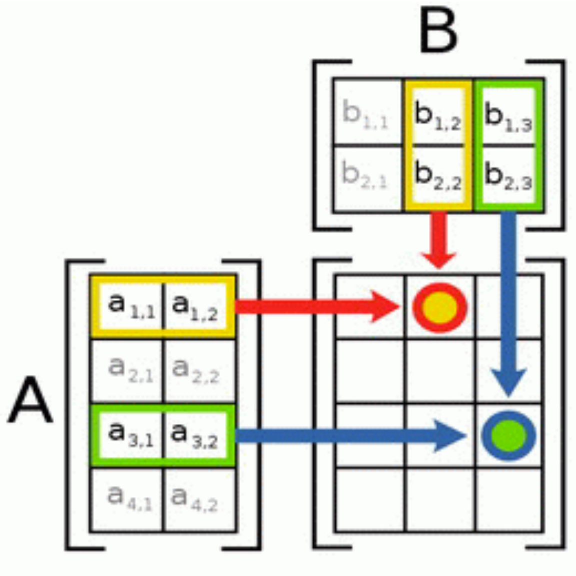
$$A\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \end{bmatrix}$$

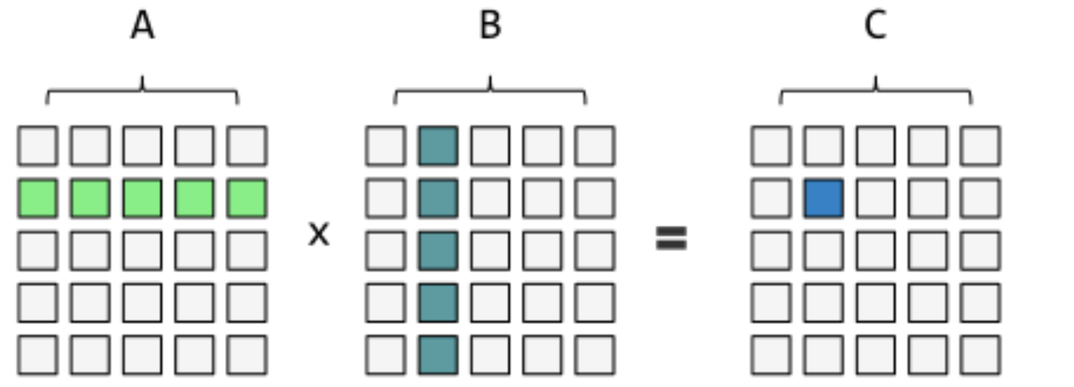
```
void matvec_base(int M, int N, REAL Y[], REAL A[][N], REAL B[]) {
    int i, j;
    for (i = 0; i < M; i++) {
        REAL temp = 0.0;
        for (j = 0; j < N; j++) {
            temp += A[i][j] * B[j];
        }
        Y[i] = temp;
    }
}
```

BLAS 3

- BLAS 3, 1987-1988
  - 9 operations (mostly) on matrix/matrix pairs
    - "GEMM": C = α·A·B + β·C, C = α·A·AT + β·C, B = T-1·B
  - Up to 4 versions of each (S/D/C/Z), 30 routines, 10K LOC
  - Why BLAS 3 ? They do $O(n^3)$ ops on $O(n^2)$ data
    - Computational intensity $(2n^3)/(4n^2) = n/2$ – big at last!
    - Good for machines with caches, deep mem hierarchy

A[M][K] * B[K][N] = C[M][N]

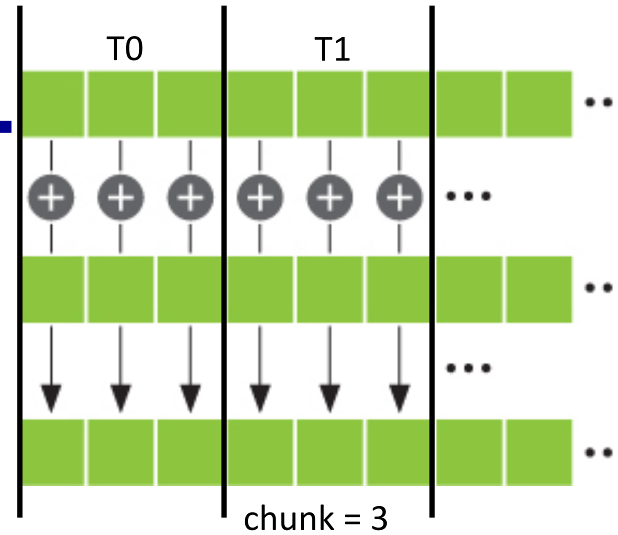




C[i][j] = sum(A[i][k] * B[k][j]) for k = 0 ... n

Decomposition for AXPY, Matrix Vector, and Matrix Multiplication

BLAS 1: AXPY

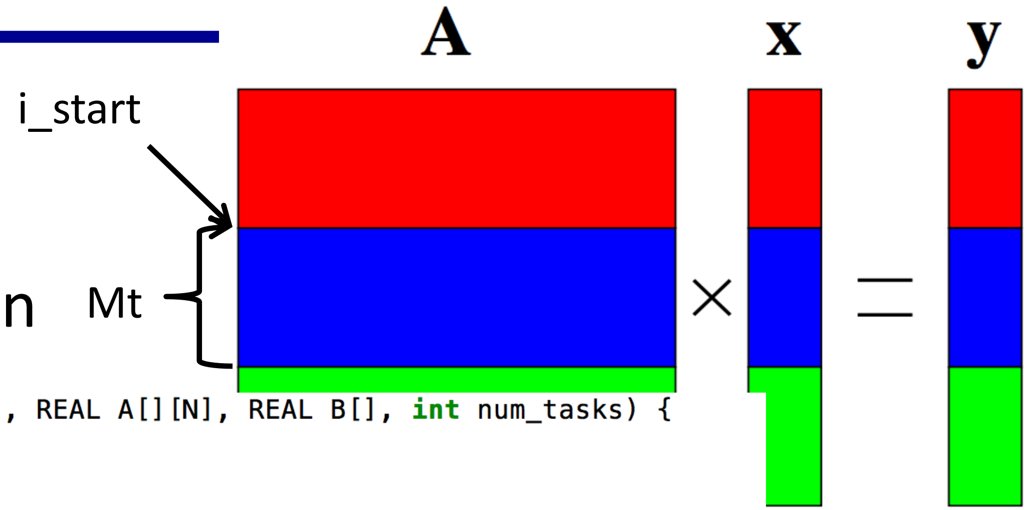


- $y = \alpha \cdot x + y$
  - x and y are vectors of size N
    - In C, x[N], y[N]
  - α is scalar
- Decomposition is simple
  - N iterations (N elements of X and Y) are distributed among threads
  - 1:1 mapping between iteration and element of X and Y
  - X and Y are shared

```c
void axpy_openmp(int N, REAL Y[], REAL X[], REAL a) {
    int i;
    #pragma omp parallel for
    for (i = 0; i < N; ++i)
        Y[i] += a * X[i];
}
```

BLAS 2: Matrix Vector Multiplication

- y = A·x
  - A[M][N], x[N], y[N]
- Row-wise decomposition

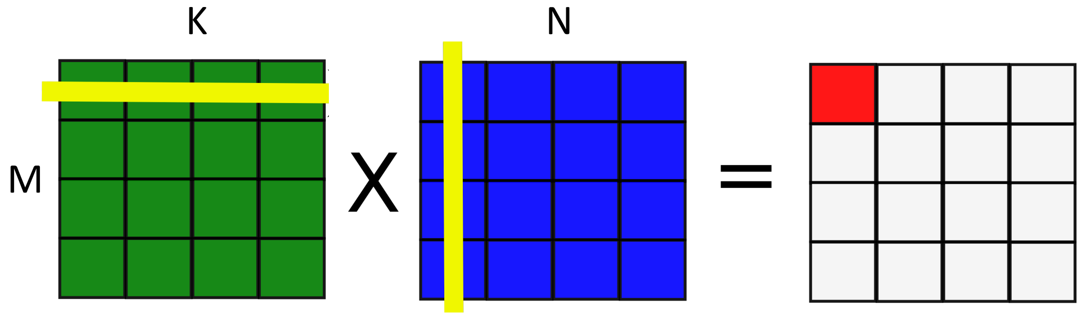



```c
void matvec_omp_parallel(int M, int N, REAL Y[], REAL A[][N], REAL B[], int num_tasks) {
    #pragma omp parallel num_threads(num_tasks)
    {
        int tid = omp_get_thread_num();
        int Mt = N/num_tasks;
        int i_start = tid*Mt;
        int i, j;
        for (i = i_start; i < i_start + Mt; i++) {
            REAL temp = 0.0;
            for (j = 0; j < N; j++) {
                temp += A[i][j] * B[j];
            }
            Y[i] = temp;
        }
    }
}

void matvec_omp_parallel_for(int M, int N, REAL Y[], REAL A[][N], REAL B[], int num_tasks) {
    int i, j;
    #pragma omp parallel for private(i,j) num_threads(num_tasks)
    for (i = 0; i < M; i++) {
        REAL temp = 0.0;
        for (j = 0; j < N; j++) {
            temp += A[i][j] * B[j];
        }
        Y[i] = temp;
    }
}
```

BLAS 3: Dense Matrix Multiplication

A[M][K] * B[k][N] = C[M][N]

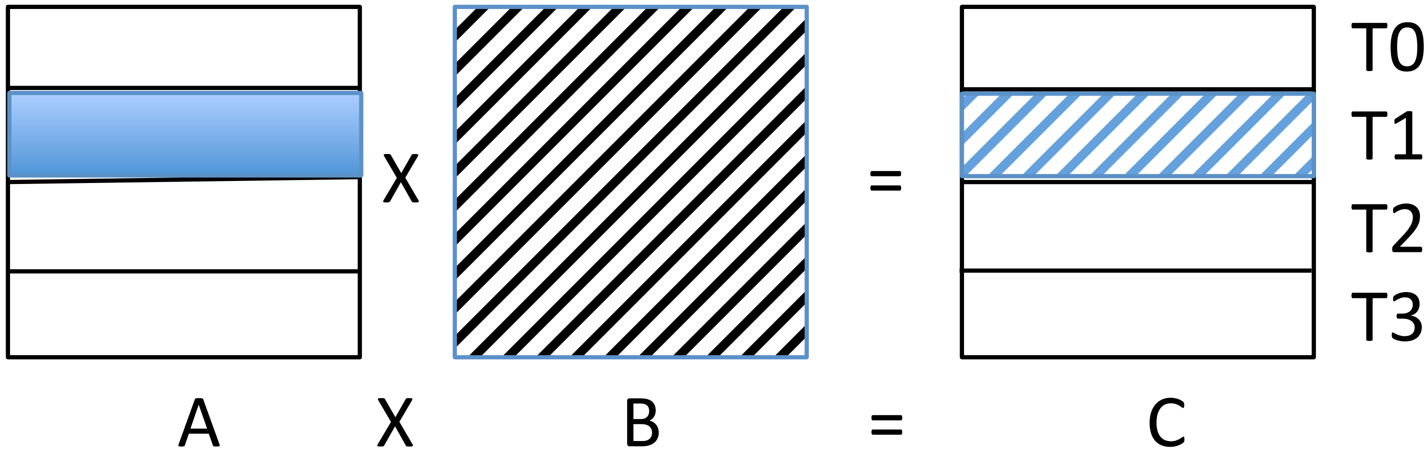
- Base
- Base_1: column major order of access
- row1D_dist
- column1D_dist
- rowcol2D_dist



- Decomposition is to calculate Mt and Nt

BLAS 3: Dense Matrix Multiplication

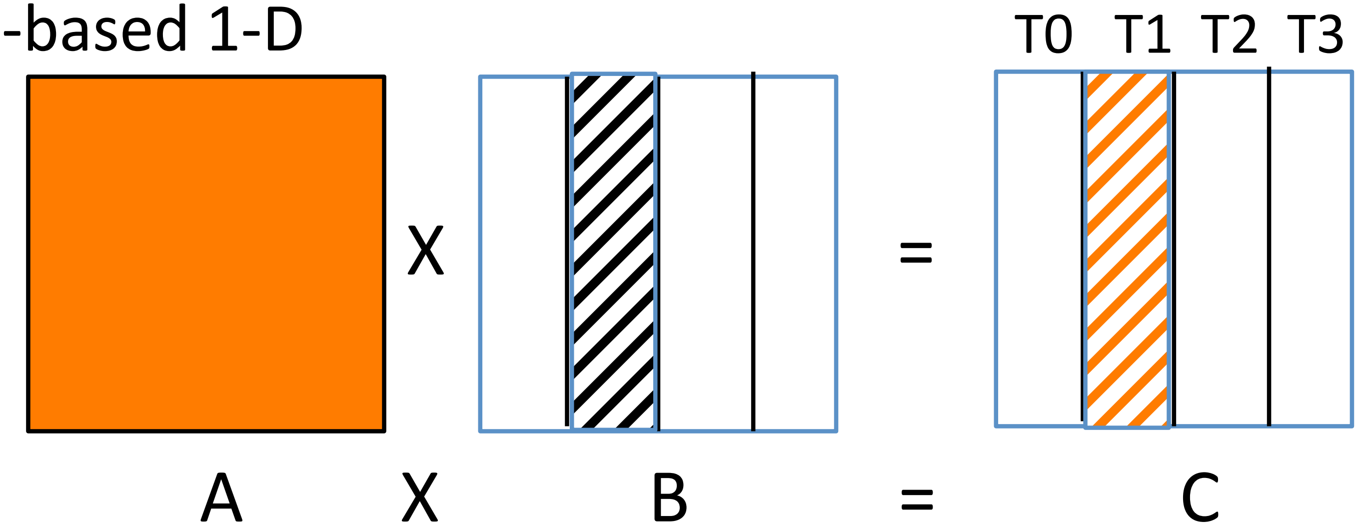
- Row-based 1-D




```c
void mm_omp_parallel_for_row1D(int N, int K, int M, REAL * A, REAL
    int i, j, w;
    #pragma omp parallel private (i, j, w) num_threads(4)
    #pragma omp for
    for (i=0; i<N; i++)
        for (j=0; j<M; j++) {
            REAL temp = 0.0;
            for (w=0; w<K; w++)
                temp += A[i*K+w]*B[w*M+j];
            C[i*M+j] = temp;
        }
}
```

BLAS 3: Dense Matrix Multiplication

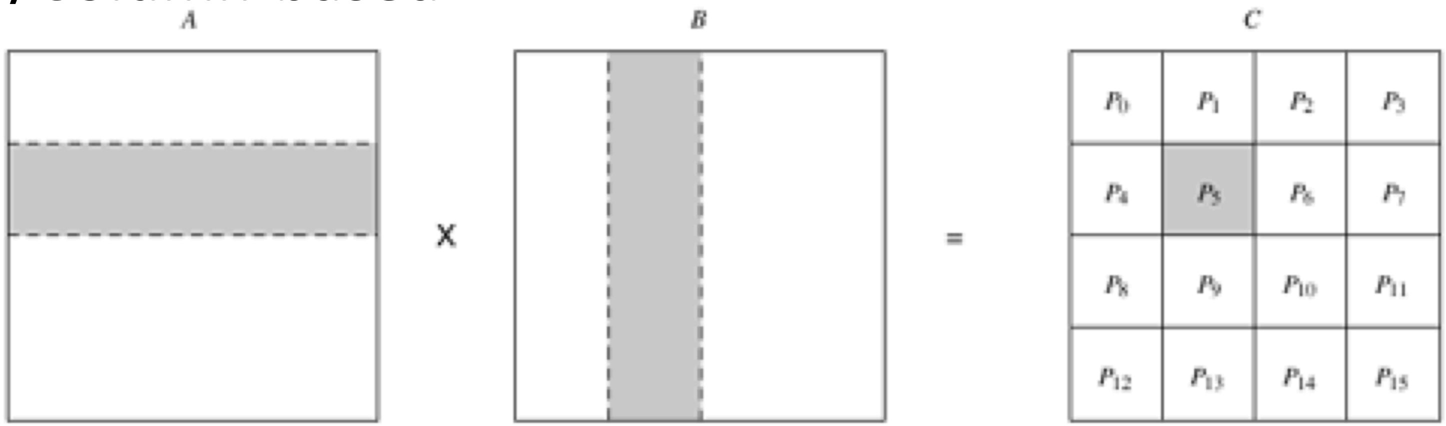
- Column-based 1-D




```c
void mm_omp_parallel_for_col1D(int N, int K, int M, REAL * A, R
    int i, j, w;
    #pragma omp parallel private (i, j, w) num_threads(4)
    for (i=0; i<N; i++)
        #pragma omp for
        for (j=0; j<M; j++) {
            REAL temp = 0.0;
            for (w=0; w<K; w++)
                temp += A[i*K+w]*B[w*M+j];
            C[i*M+j] = temp;
        }
}
```

BLAS 3: Dense Matrix Multiplication

- Row/Column-based 2-D




```
void mm_omp_parallel_for_rowcol2D(int N, int K, int M, REAL * A, REAL *
    int i, j, w;
    #pragma omp parallel for private (i, j, w) num_threads(4)
    for (i=0; i<N; i++)
        #pragma omp parallel for shared(i) private (j, w) num_threads(4)
        for (j=0; j<M; j++) {
            REAL temp = 0.0;
            for (w=0; w<K; w++)
                temp += A[i*K+w]*B[w*M+j];
            C[i*M+j] = temp;
        }
}
```

Need nested parallelism

export OMP_NESTED=true

Dense matrix algorithms

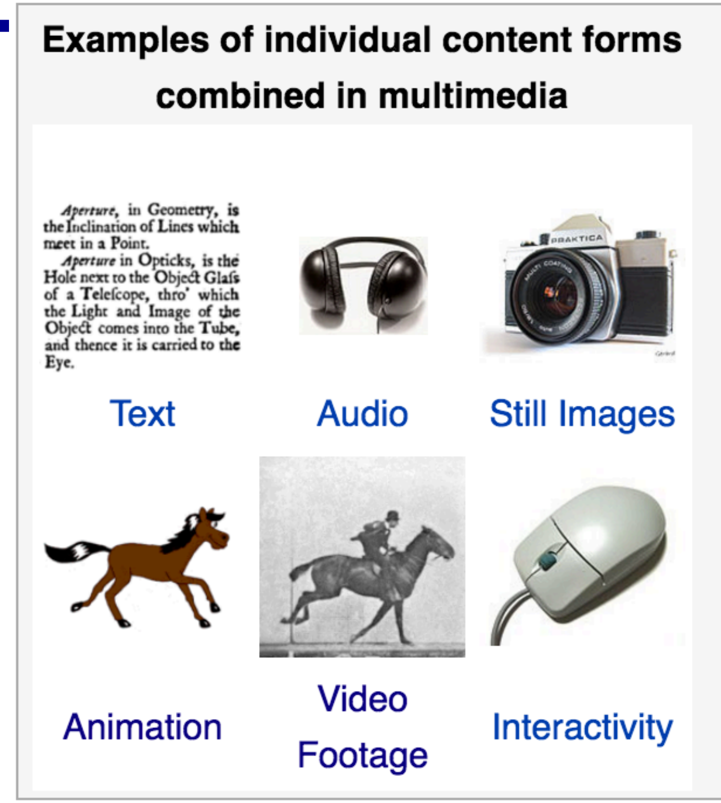
- **Dense linear algebra and BLAS**
- **Image processing/stencil**
- Iterative methods

What is Multimedia

- Multimedia is a combination of text, graphic, sound, animation, and video that is delivered interactively to the user by electronic or digitally manipulated means.

Medium	Elements	Time-dependence
Text	Printable characters	No
Graphic	Vectors, regions	No
Image	Pixels	No
Audio	Sound, Volume	Yes
Video	Raster images, graphics	Yes

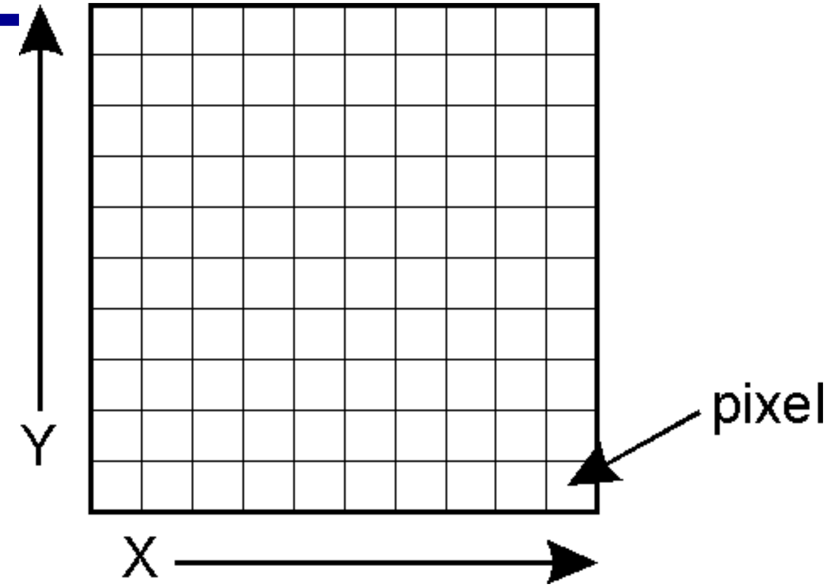
Videos contains frame (images)



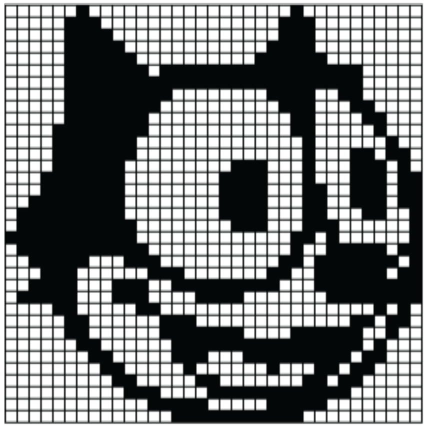
https://en.wikipedia.org/wiki/Multimedia

Image Format and Processing

- Pixels
  - Images are matrix of pixels



- Binary images
  - Each pixel is either 0 or 1



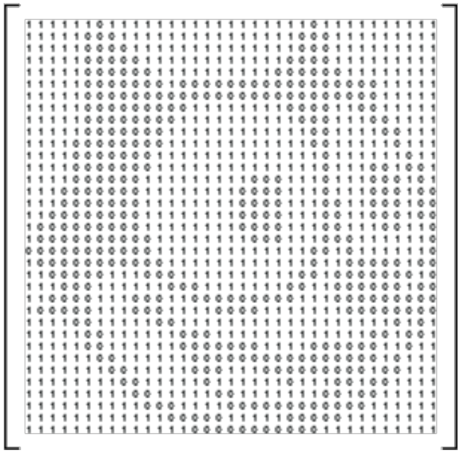
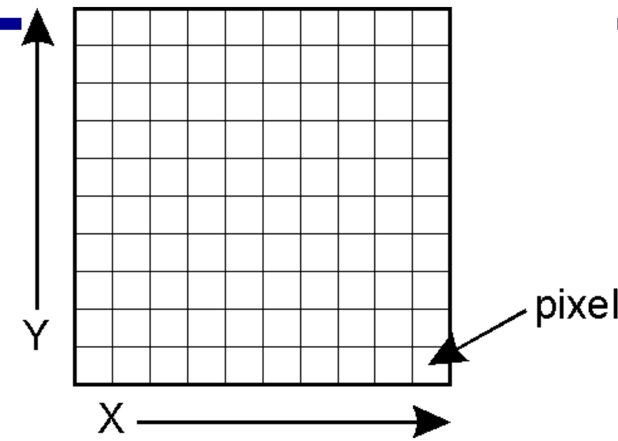


Image Format and Processing

- Pixels
  - Images are matrix of pixels

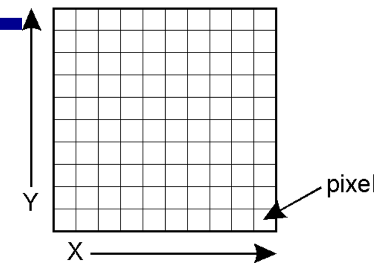


- Grayscale images
  - Each pixel value normally range from 0 (black) to 255 (white)
  - 8 bits per pixel

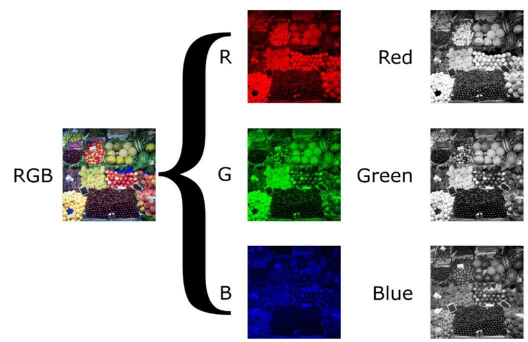
But the camera sees this:

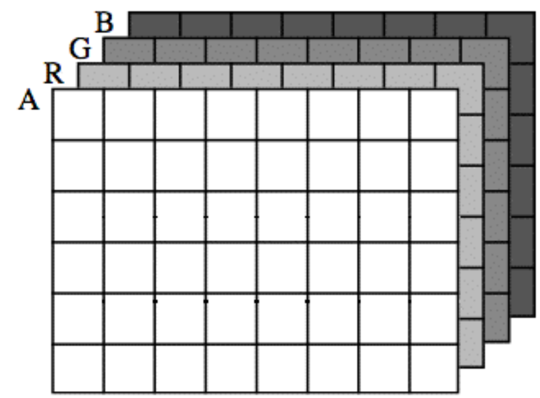
194	210	201	212	199	213	215	195	178	158	182	209
180	189	190	221	209	205	191	167	147	115	129	163
114	126	140	188	176	165	152	140	170	106	78	88
87	103	115	154	143	142	149	153	173	101	57	57
102	112	106	131	122	138	152	147	128	84	58	66
94	95	79	104	105	124	129	113	107	87	69	67
68	71	69	98	89	92	98	95	89	88	76	67
41	56	68	99	63	45	60	82	58	76	74	65
20	41	69	75	56	41	51	73	55	70	63	44
50	50	57	69	75	75	73	74	53	68	59	37
72	59	53	66	84	92	84	74	57	72	63	42
67	61	58	65	75	78	76	73	59	75	69	50

Image Format and Processing



- Pixels
  - Images are matrix of pixels
- Color images
  - Each pixel has three/four values (4 bits or 8 bits each) each representing a color scale



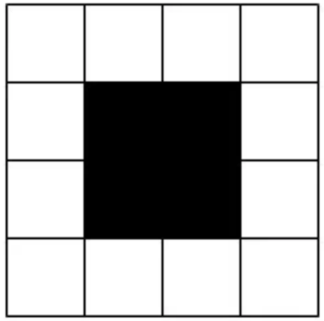


Sample Length:	4	4	4	4
Channel Membership:	Alpha	Red	Green	Blue
Bit Number:	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0

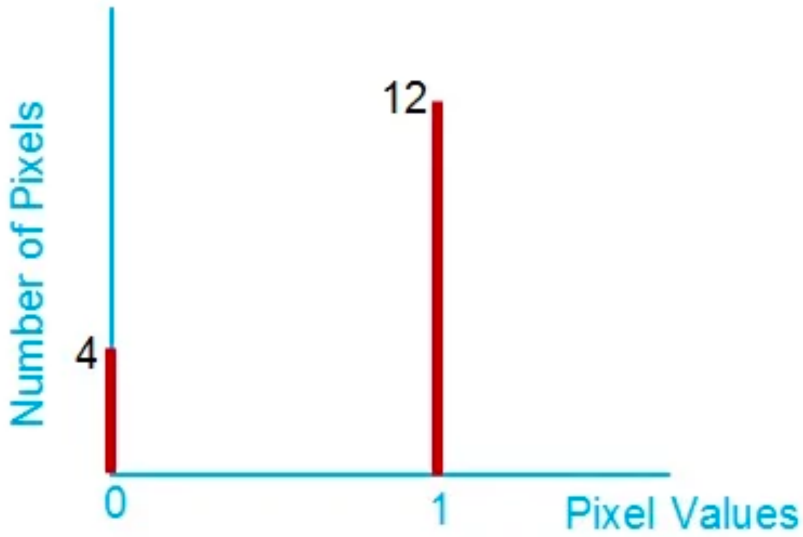
Sample Length:	8	8	8	8
Channel Membership:	Blue	Green	Red	Alpha
Bit Number:	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0

Histogram

- An image histogram is a graph of pixel intensity (on the *x*-axis) versus number of pixels (on the *y*-axis). The *x*-axis has all available gray levels, and the *y*-axis indicates the number of pixels that have a particular gray-level value.



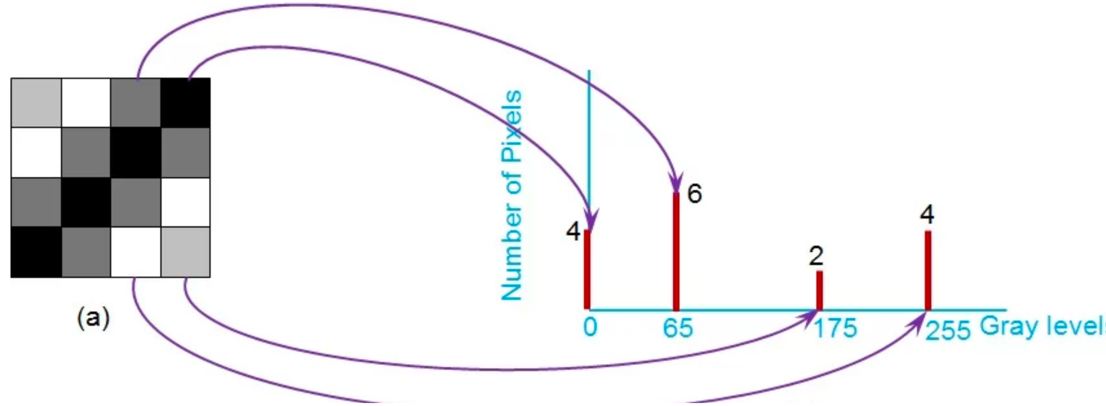
(a)



(b)

https://www.allaboutcircuits.com/technical-articles/image-histogram-characteristics-machine-learning-image-processing/

Histograms of Monochrome Image




```
calculate_histogram(image, histogram, length, width)
   int     length, width;
   short   **image;
   unsigned long histogram[];
{
   long  i,j;
   short k;
   for(i=0; i<length; i++){
      for(j=0; j<width; j++){
         k = image[i][j];
         histogram[k] = histogram[k] + 1;
      }
   }
}  /* ends calculate_histogram */
```

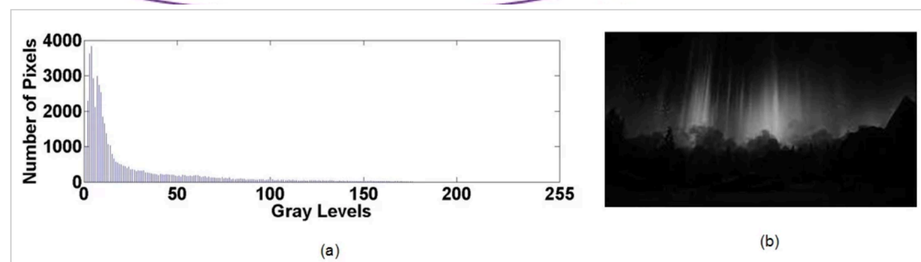


Figure 5. *Histogram of a dark image. Image by Sneha H.L.*

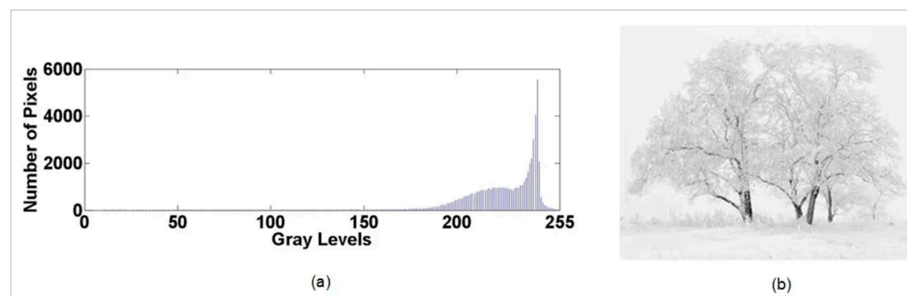
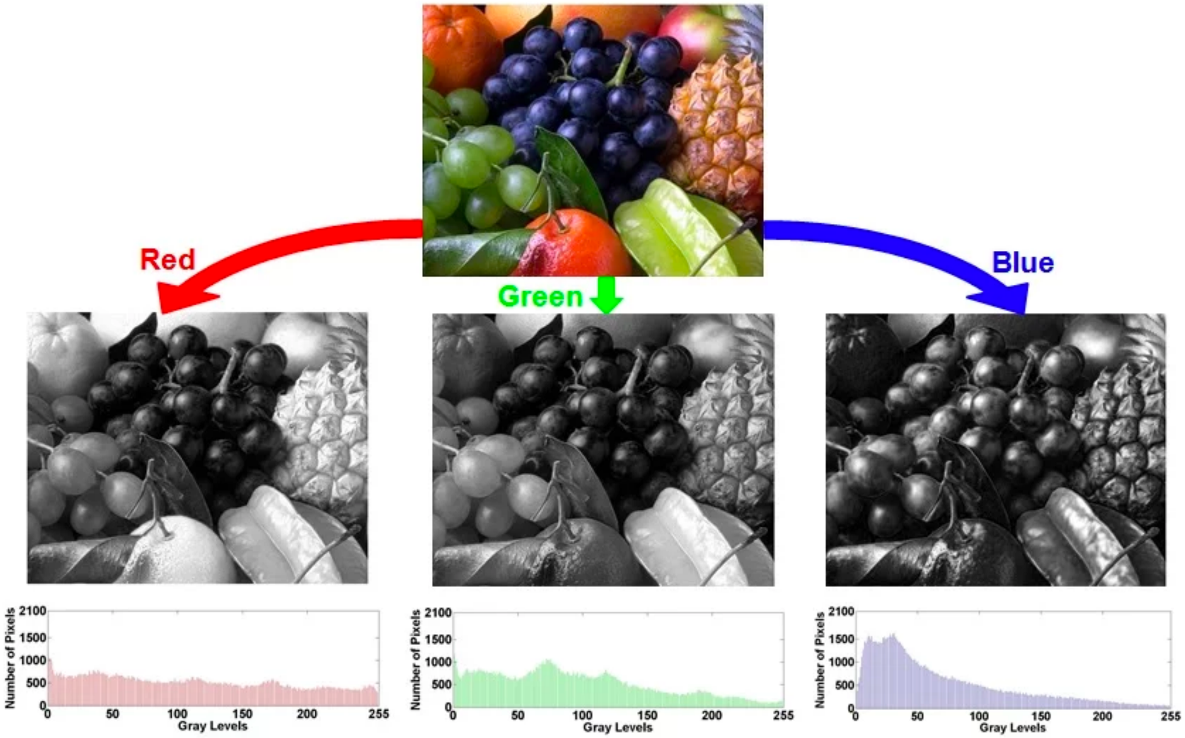


Figure 6. *Histogram of a bright image. Image by Sneha H.L.*

http://homepages.inf.ed.ac.uk/rbf/BOOKS/PHILLIPS/cips2edsrc/HIST.C

Histogram of Color Images

- Image density




```
for( int y = 0; y < image.rows; y++ ) {
    for( int x = 0; x < image.cols; x++ ) {
        for( int c = 0; c < 3; c++ ) {
            new_image.at<Vec3b>(y,x)[c] =
              saturate_cast<uchar>( alpha*( image.at<Vec3b>(y,x)[c] ) + beta );
        }
    }
}
```

https://docs.opencv.org/3.4.0/d3/dc1/tutorial_basic_linear_transform.html

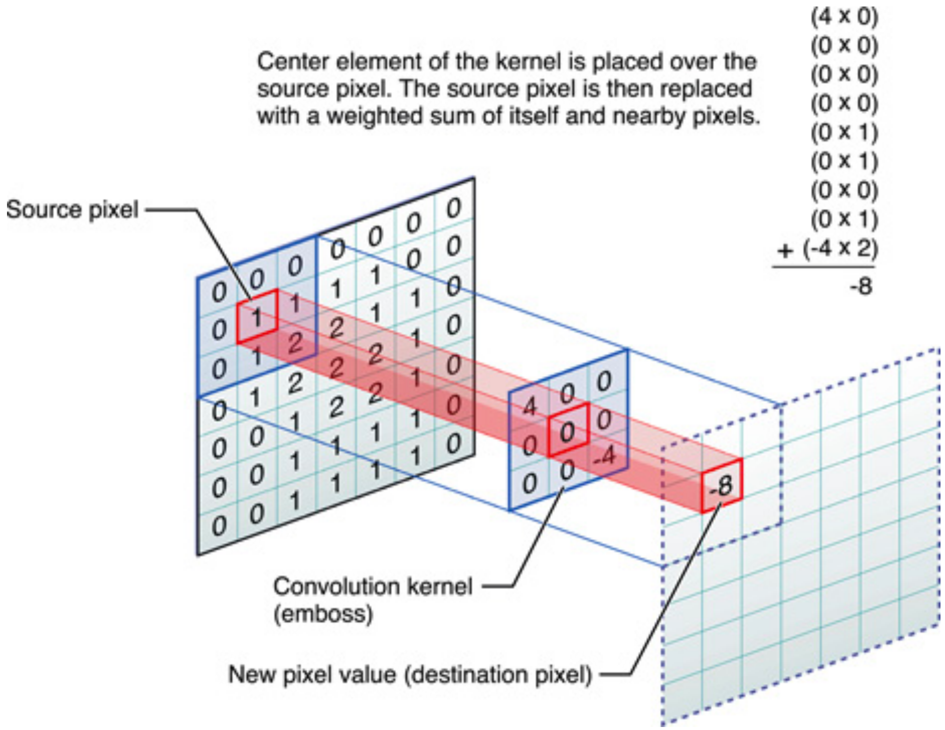
OpenMP Parallelization of Histogram

- Decomposition based on output (pixel values, 0 - 255)
  - Each thread **searches the whole image** to only count those pixels that have the value it should count for
    - E.g. with 4 threads: 0-63 for thread 0, 64-127 for thread 1, …

- Decomposition based on the input (image)
  - Each thread **search part of the image to count all the pixels** and store the **partial** histogram **locally**
  - Add up all the partial histogram

Image Filtering

- Changing pixel values by doing a [convolution]() between a kernel (filter) and an image.




```
for(i=1; i<rows-1; i++){
   if( (i%10) == 0) printf("%d ", i);
   for(j=1; j<cols-1; j++){
      sum = 0;
      for(a=-1; a<2; a++){
         for(b=-1; b<2; b++){
            sum = sum +
                  the_image[i+a][j+b] *
                  filter[a+1][b+1];
         }
      }
      sum                 = sum/d;
      if(sum < 0)    sum = 0;
      if(sum > max) sum = max;
      out_image[i][j]     = sum;

   }  /* ends loop over j */
}  /* ends loop over i */
```

Image Filtering: The magic of the filter matrix

- http://lodev.org/cgtutor/filtering.html
- https://en.wikipedia.org/wiki/Kernel_(image_processing)

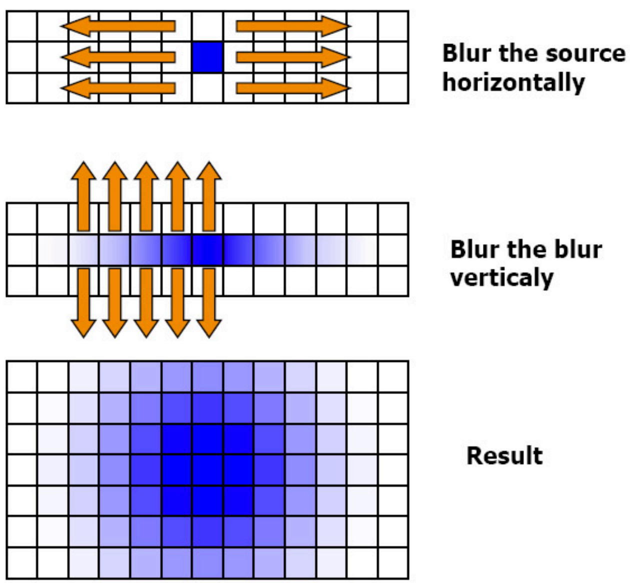
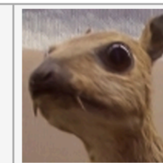
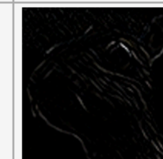
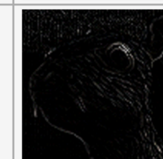
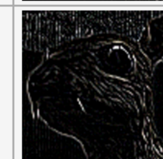
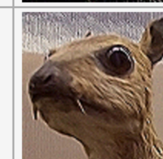
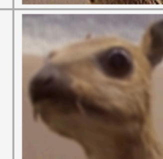
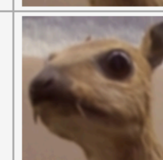
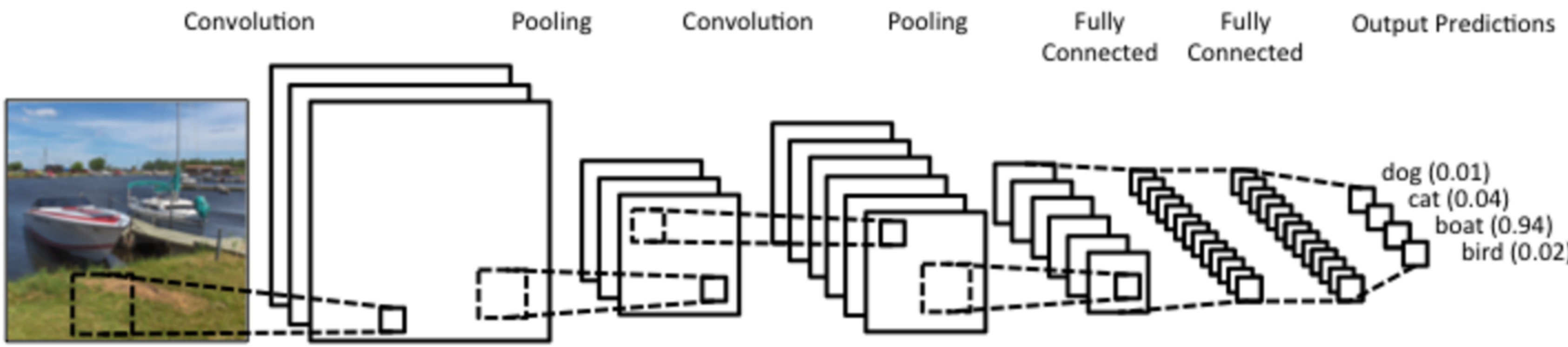


Image taken from ATI's presentation

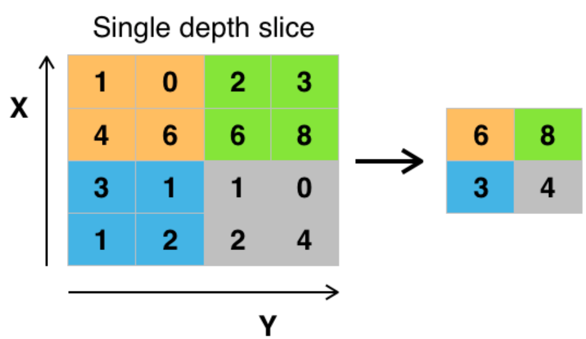
- **It is the basic of convolution neural network**

Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Convolution Neural Network for Object Detection



- **Pooling: sample-based discretization process**

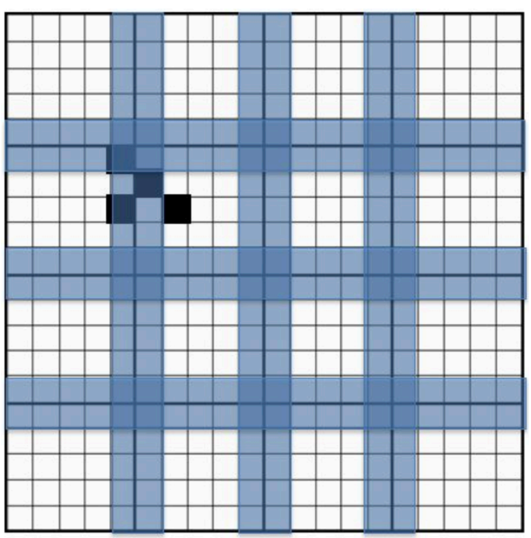


Example of Maxpool with a 2x2 filter and a stride of 2

http://cs231n.github.io/convolutional-networks/

OpenMP Parallelization of Image Filtering

- Decomposition according to the input image
- Since input and output images are separate, it is straightforward
  - Could be row1D, col1D, rowcol2D
- False-sharing for writing boundary of output images



```
for(i=1; i<rows-1; i++){
   if( (i%10) == 0) printf("%d ", i);
   for(j=1; j<cols-1; j++){
      sum = 0;
      for(a=-1; a<2; a++){
         for(b=-1; b<2; b++){
            sum = sum +
                  the_image[i+a][j+b] *
                  filter[a+1][b+1];
         }
      }
      sum                 = sum/d;
      if(sum < 0)    sum = 0;
      if(sum > max) sum = max;
      out_image[i][j]   = sum;

   }  /* ends loop over j */
}  /* ends loop over i */
```

Dense matrix algorithms

- Dense linear algebra and BLAS
- Image processing/stencil
- Iterative methods

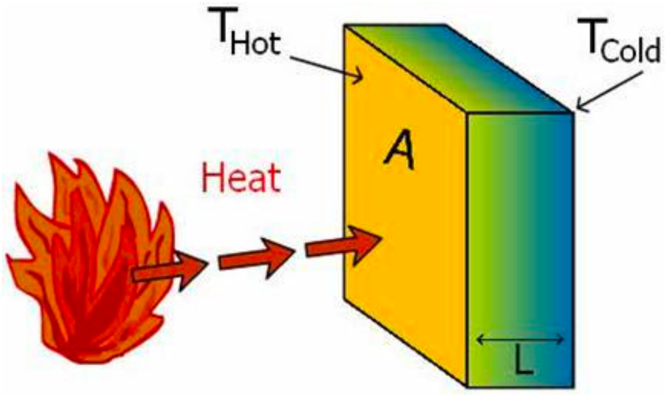
Iterative Methods

- **Iterative methods can be expressed in the general form:**

$$x^{(k)} = F(x^{(k-1)})$$

Hopefully: $x^{(k)} \to s$ (solution of my problem)

- **Wide variety of computational science problem**
  - CFD, molecular dynamics, weather/climate forecast, cosmology,
- Will it converge? How rapidly?

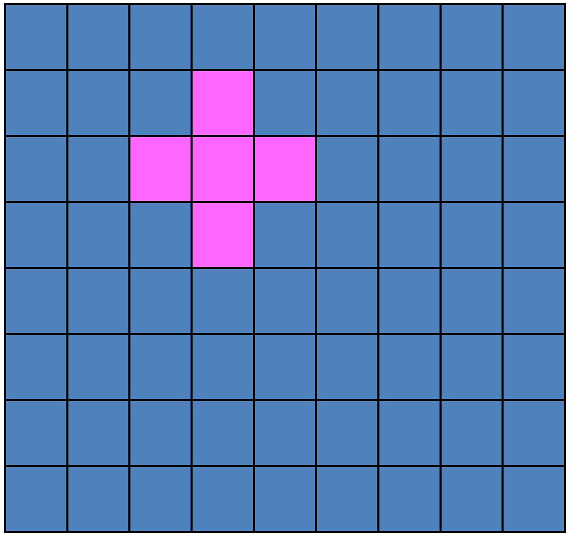


Iterative Stencil Applications

Loop until some condition is true

$x^{(k)} = F(x^{(k-1)})$

Perform computation which involves communicating with N,E,W,S neighbors of a point (5 point stencil)

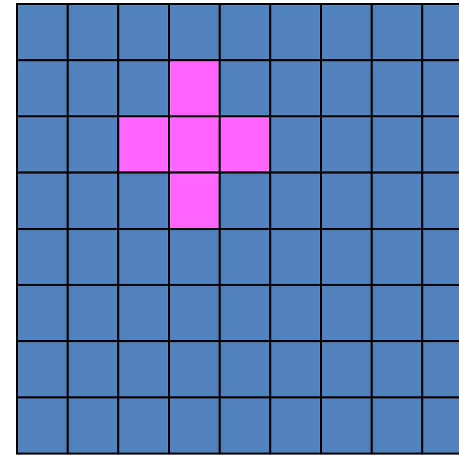


[Convergence test?]

Stencil is similar as image filtering/convolution

Jacobi.c

- Assignment 2 and 3:



```c
  while ((k <= mits) && (error > tol)) {
    error = 0.0;

    /* Copy new solution into old */
    for (i = 0; i < n; i++)
      for (j = 0; j < m; j++)
        uold[i][j] = u[i][j];

    for (i = 1; i < (n - 1); i++)
      for (j = 1; j < (m - 1); j++) {
        resid = (ax * (uold[i - 1][j] + uold[i + 1][j]) +
                ay * (uold[i][j - 1] + uold[i][j + 1]) +
                b * uold[i][j] - f[i][j]) / b;
        //printf("i: %d, j: %d, resid: %f\n", i, j, resid);

        u[i][j] = uold[i][j] - omega * resid;
        error = error + resid * resid;
      }
    /* Error check */
    if (k % 500 == 0)
    printf("Finished %ld iteration with error: %g\n", k, error);
    error = sqrt(error) / (n * m);

    k = k + 1;
  } /*  End iteration loop */
  printf("Total Number of Iterations: %ld\n", k);
  printf("Residual: %.15g\n", error);
}
```

https://passlab.github.io/CSCE569/Assignment_2/jacobi.c

Jacobi

- An iterative method for approximating the solution to a system of linear equations.
- ***Ax=b*** where the ith equation is

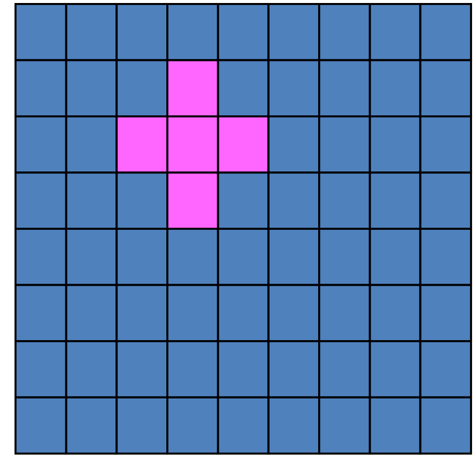
$$a_{i,1}x_1 + a_{i,1}x_1 + \cdots + a_{i,n}x_n = b_i$$

$$x_i = \frac{1}{a_{i,i}}\left[b_i - \sum_{j \neq i} a_{i,j}x_j\right]$$

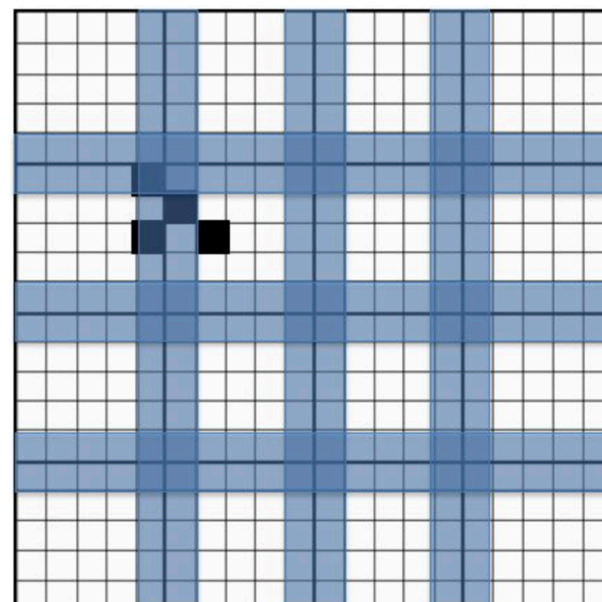
- *a*'s and *b*'s are known, **want to solve for *x*'s**

OpenMP Parallelization of Jacobi

- Similar as image filtering
  - Enclosed by the *while* to be iterative
- omp parallel for outer *while* loop
- omp for for inner for loops
- single and reduction are needed

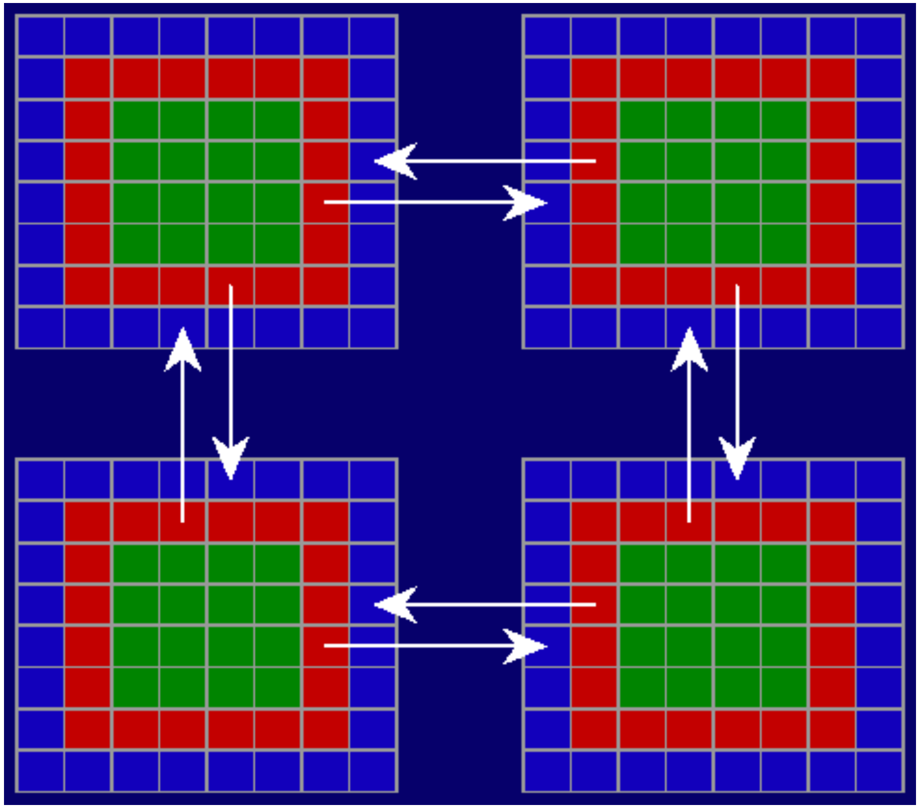


```c
  while ((k <= mits) && (error > tol)) {
    error = 0.0;

    /* Copy new solution into old */
    for (i = 0; i < n; i++)
      for (j = 0; j < m; j++)
        uold[i][j] = u[i][j];

    for (i = 1; i < (n - 1); i++)
      for (j = 1; j < (m - 1); j++) {
        resid = (ax * (uold[i - 1][j] + uold[i + 1][j]) +
                ay * (uold[i][j - 1] + uold[i][j + 1]) +
                b * uold[i][j] - f[i][j]) / b;
        //printf("i: %d, j: %d, resid: %f\n", i, j, resid);

        u[i][j] = uold[i][j] - omega * resid;
        error = error + resid * resid;
      }
    /* Error check */
    if (k % 500 == 0)
    printf("Finished %ld iteration with error: %g\n", k, error);
    error = sqrt(error) / (n * m);

    k = k + 1;
  } /*  End iteration loop */
  printf("Total Number of Iterations: %ld\n", k);
  printf("Residual: %.15g\n", error);
}
```



Ghost Cell Exchange

- For assignment 3:

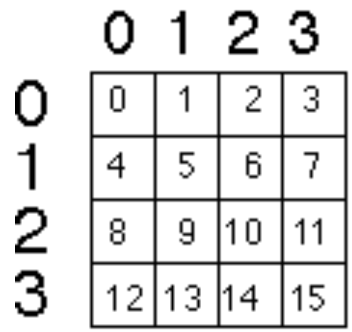


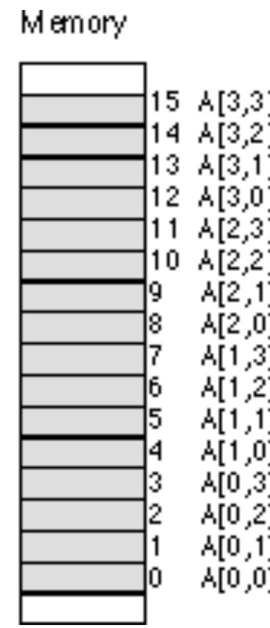
Background:
C multidimensional array

Vector/Matrix and Array in C

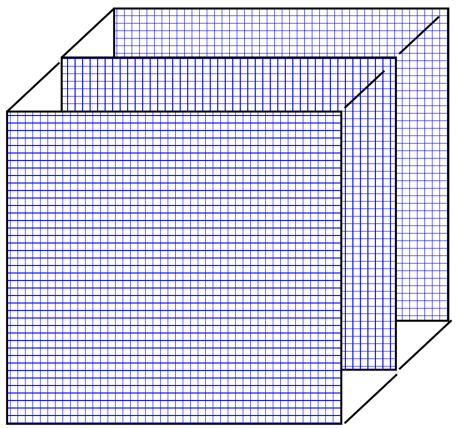
- C has row-major storage for multiple dimensional array
  - A[2,2] is followed by A[2,3]

char A[4][4]





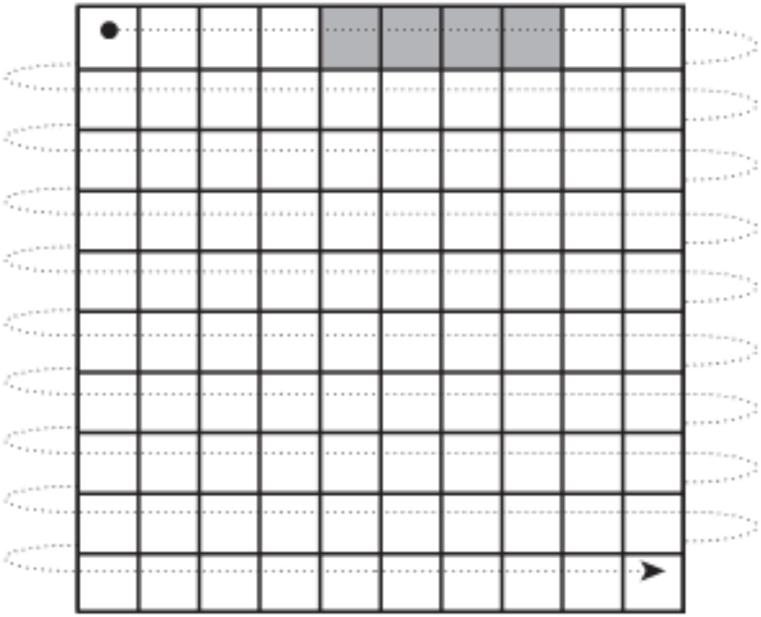
- 3-dimensional array
  - B[3][100][100]



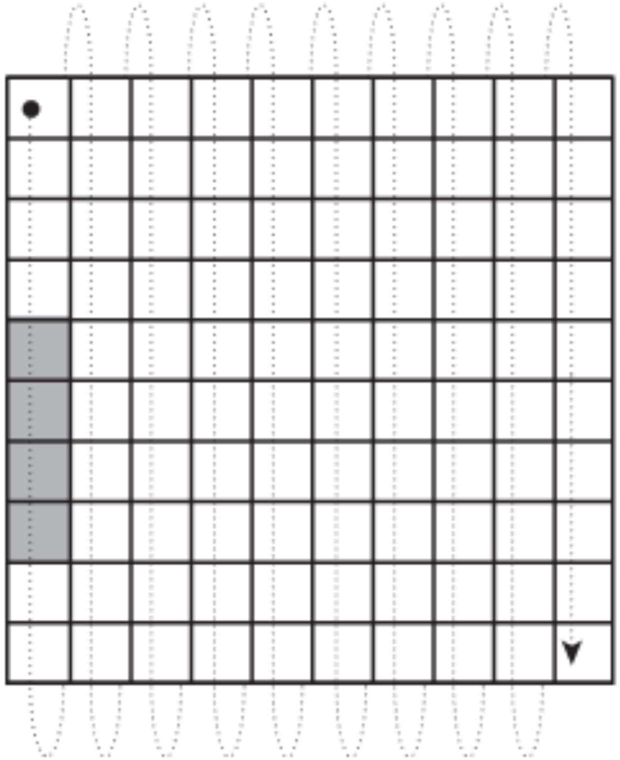
- Think it as recursive definition
  - A[4][10][32]

Column Major

Fortran is column major



Row-major order



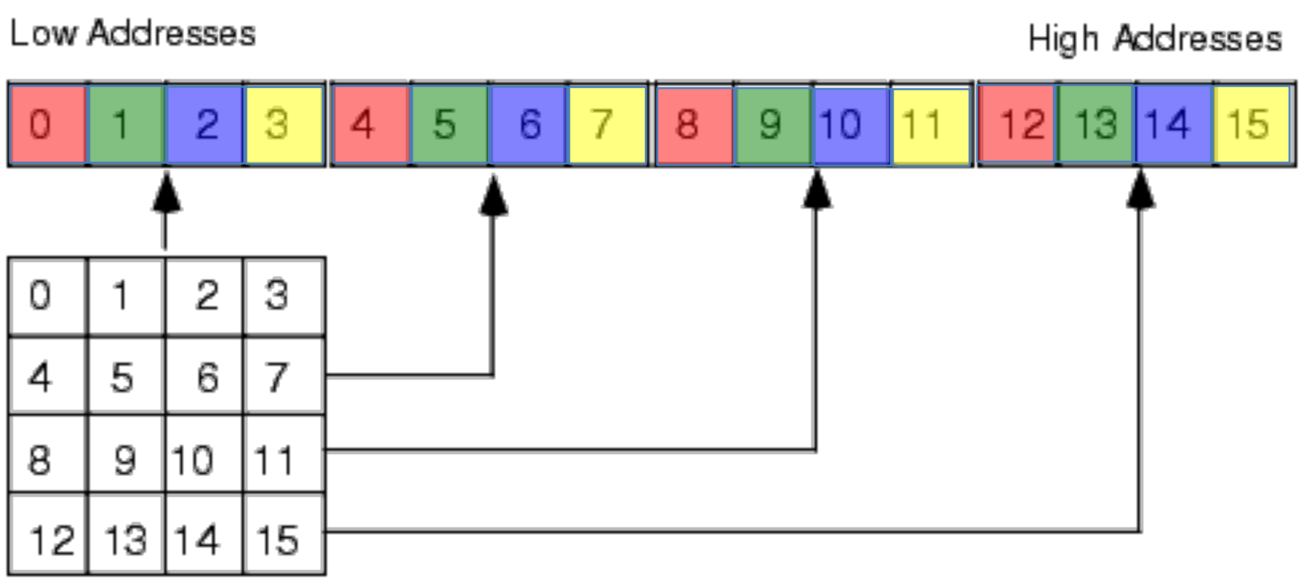
Column-major order

Array Layout: Why We Care?

1. Makes a big difference for access speed

- For performance, set up code to go in row major order in C
  - Caching: each read from memory will bring other adjacent elements to the cache line
- (Bad) Example: 4 vs 16 accesses
  - matmul_base_1

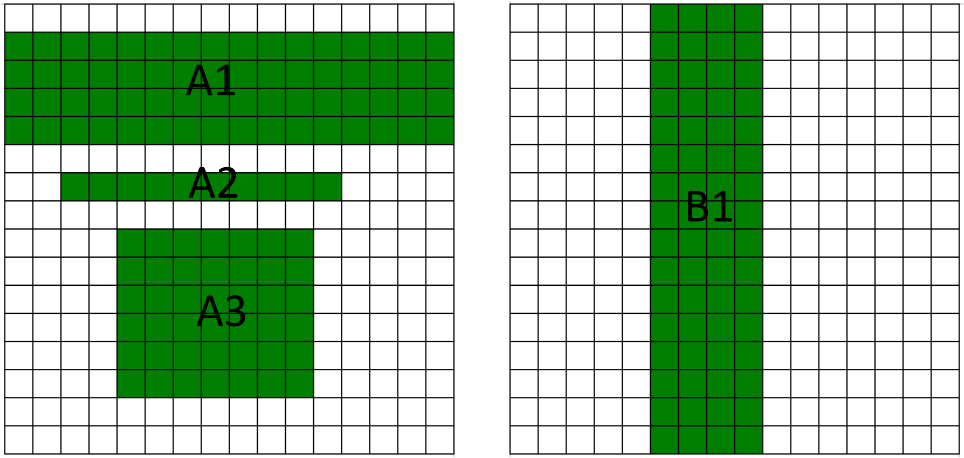
```
for i = 1 to n
        for j = 1 to n
                A[j][i] = value
```



Array Layout: Why We Care?

2. Affect decomposition and data movement

- Decomposition may create submatrices that are in non-contiguous memory locations, e.g. A3 and B1
- Submatrices in contiguous memory location of 2-D row major matrix
  - A single-row submatrix, e.g. A2
  - A submatrix formed with adjacent rows with full column length, e.g. A1

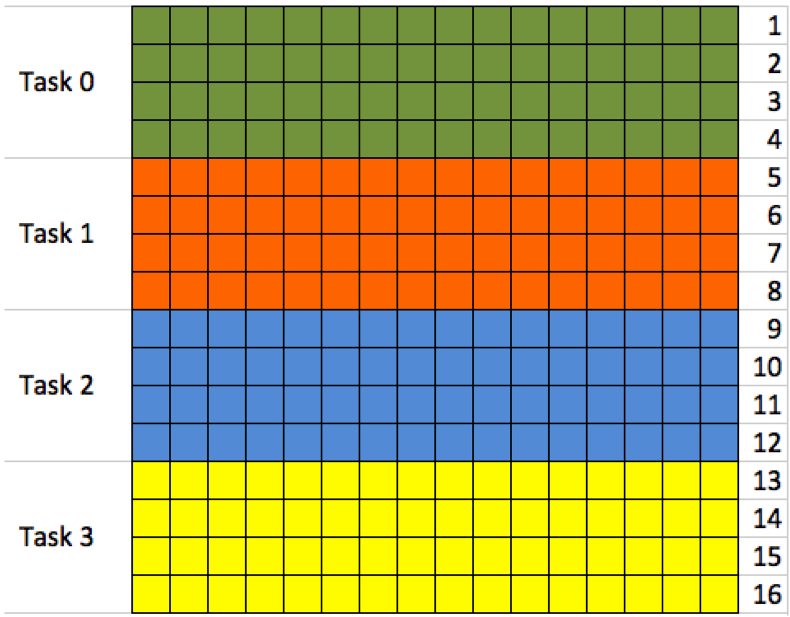


Array Layout: Why We Care?

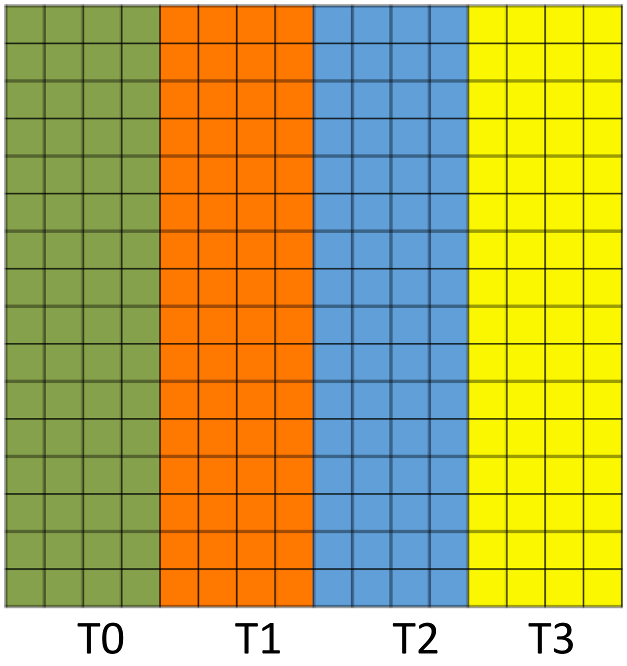
2. Affect decomposition and submatrix

- Row or column wise distribution of 2-D row-major array
- # of data movement to exchange data between T0 and T1
  - Row-wise: one memory copy by each
  - Column-wise: 16 copies each

Row-wise distribution



Column-wise distribution



Array and pointers in C

- In C, an array is a pointer + dimensionality
  – They are literally the same in binary, i.e. pointer to the first element, referenced as base address
- Cast and assignment from array to pointe, int A[M][N]
  - A, &A[0][0], and A[0] have the same value, i.e. the pointer to the first element of the array
- Cast a pointer to an array
  – int *ap; int (*A)[N] = (int(*)[N])ap; A[i][j] ....
- Address calculation for array references
  – Address of A[i][j] = A + (i*N+j)*sizeof (int)