# Numerical Linear Algebra for Computational Science and Information Engineering

## Lecture 17
## Introduction to Communication-Avoiding Algorithms

Nicolas Venkovic
nicolas.venkovic@tum.de

Group of Computational Mathematics
School of Computation, Information and Technology
Technical University of Munich

Summer 2025

# Outline

# Introduction

CS294/Math270 – Demmel and Grigori (2016)
MIT 6.172 – Shun (2018)

# Cost of algorithm deployment

- Deploying an algorithm has **two costs** measured in **time** (or **energy**):
  - **Arithmetic** (floating-point operations, FLOPs)

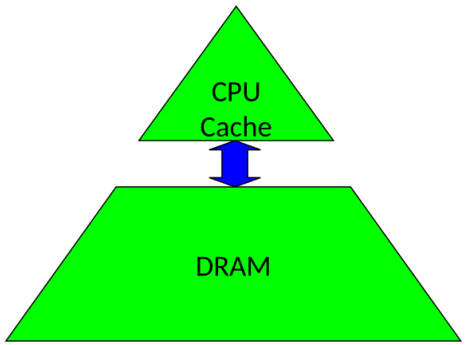
$$\#\text{ of }\textbf{FLOPs} \div (\#\text{ of }\textbf{FLOPs per cycle} \times \#\textbf{ cycles per unit of time})$$

  - **Communication** due to data movement between
    - **levels of a memory hierarchy**:

      **sequential** part of a program

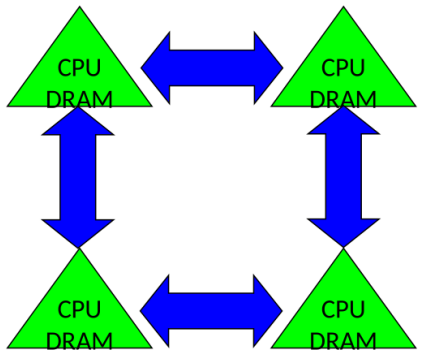
      transfers between DRAM, L3 cache, L2 cache, L1 cache and registers

      cache hit, cache miss, ...




    - **processors over a network**:

      **parallel** part of a program




$$\#\text{ of }\textbf{messages} \times \textbf{latency} + \#\text{ of }\textbf{words} \div \textbf{bandwidth}$$

**idle** time

# Roofline model

- **Algorithm** on given **hardware** characterized by **arithmetic intensity**:

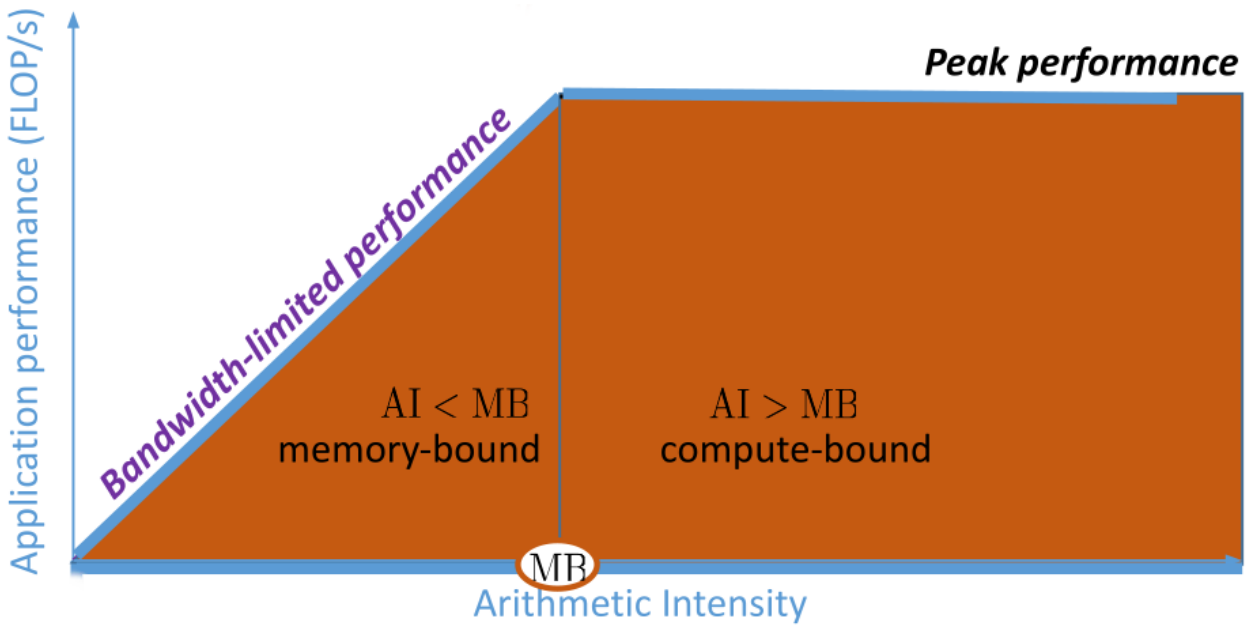
## Arithmetic intensity (AI)

Ratio of number of FLOPs over amount of data moved.

- **Hardware** characterized by **machine balance**:

## Machine balance (MB)

Ratio of peak floating-point performance over peak memory bandwidth.

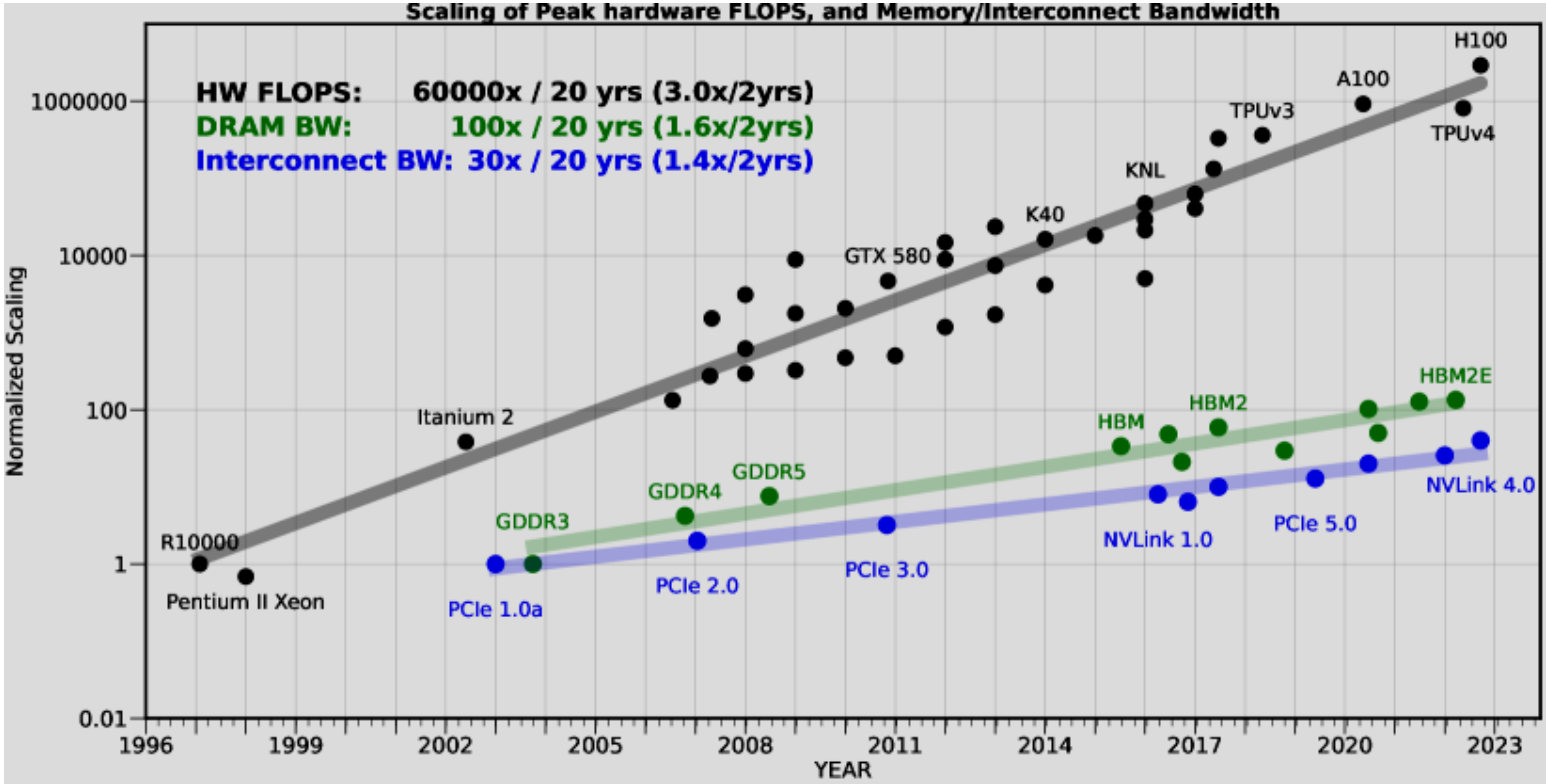


# Memory wall

- Cost to move data much greater than cost of arithmetic:

$$\textbf{time per FLOP} \ll \frac{1}{\textbf{bandwidth}} \ll \textbf{latency}$$

- Peak floating-point performance evolves faster than memory bandwidth:

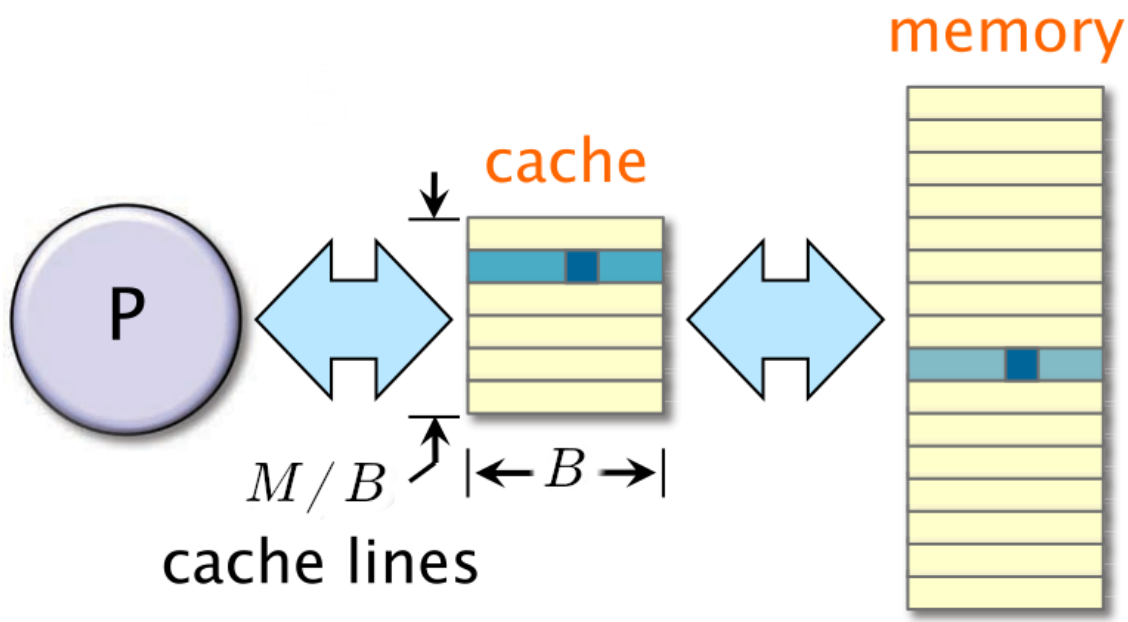



- Relative cost of algorithms due to communication larger every year

Gholami, A., Yao, Z., Kim, S., Hooper, C., Mahoney, M.W., Keutzer, K. (2024). AI and Memory Wall. arXiv:2403.14123v1.

# Ideal cache model

- An **ideal cache model** is introduced for the analysis of algorithms:




- Model parameters:
  - Two-level hierarchy
  - Cache size of $M$ bytes
  - Cache line length of $B$ bytes
  - Fully associative cache (cache lines can be stored anywhere in cache)
  - Least-recently used (LRU) cache replacement policy
  - Arbitrary large main memory

# Tall caches

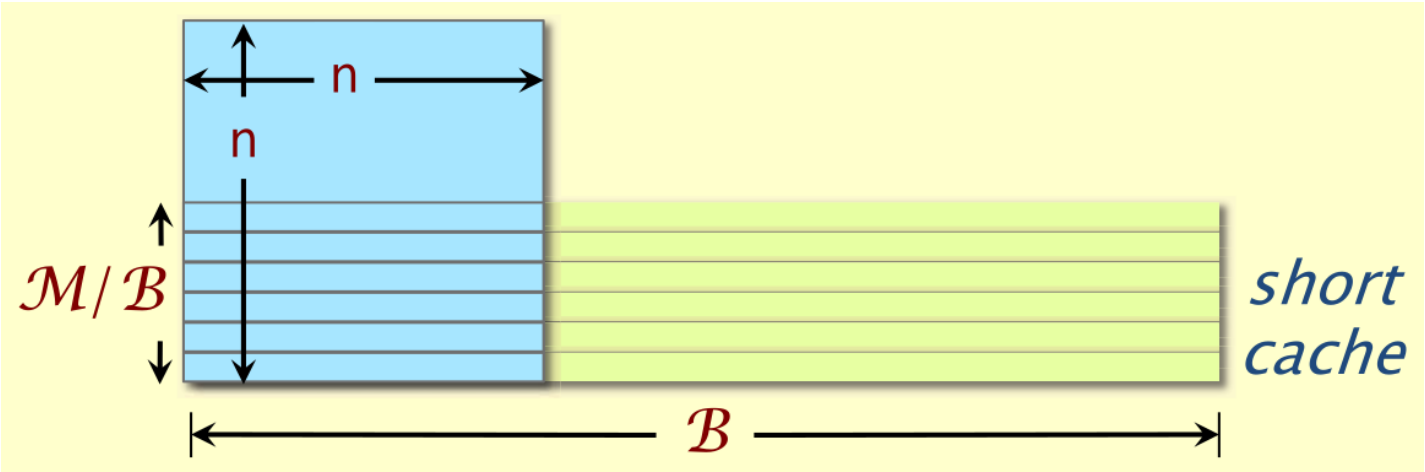
- Several of the coming cache analyses assume a tall cache:

## Tall cache assumption

A cache of size $M$ bytes with $N/B$ lines of $B$ bytes is **tall** if $B^2 < cM$ for some sufficiently small $c < 1$.

Problem of **short cache**:

- a $n \times n$ **sub**matrix may not fit in cache, even if $n^2 < cM$




i.e., cache lines store only contiguous data, and consecutive rows of a row major submatrix are not contiguous in memory.

# Communication lower bounds

- Assuming a fast memory of size $M$, Ballard et al. (2011) derive **lower bounds** on communication for valid all methods of **direct linear algebra**, **dense** or **sparse**:

$$\textbf{bandwidth cost}: \ \#\text{ of words moved} \in \Omega\left(\frac{\#\text{ of arithmetic operations}}{\sqrt{M}}\right)$$

$$\textbf{latency cost}: \ \#\text{ of messages sent} \in \Omega\left(\frac{\#\text{ of arithmetic operations}}{\sqrt{M^3}}\right)$$

- These bounds hold for
  - Matrix-multiply, LU, QR, eigensolving, SVD, tensor contraction, ...
  - Some whole programs (sequences of these operations, no matter how individual operations are interleaved, e.g. $A^k$)
  - **Sequential** and **parallel** algorithms
  - Some graph-theoretic algorithms (e.g. Floyd-Warshall)
- Significant efforts by the NLA community to rethink algorithms to try and achieve these bounds.

Ballard, G., Demmel, J., Holtz, O., & Schwartz, O. (2011). Minimizing communication in numerical linear algebra. SIAM Journal on Matrix Analysis and Applications, 32(3), 866-901.

# Matrix-matrix multiplication

MIT 6.172 – Shun (2018)

# Cache misses analysis

- Matrix-matrix multiplication ($\texttt{C} = \texttt{C} + \texttt{AB}$) with row major data:

```
for i = 0, ..., n-1                                   //zero-based indexing
  for j = 0, ..., n-1
    for k = 0, ..., n-1
      C[i*n+j] := C[i*n+j] + A[i*n+k] * B[k*n+j]
```

Cache miss complexity, $Q(n)$, **dominated by access to** $B$.

Assume a **tall** and **ideal cache**. There are 3 cases:

- Case 1: $n > cM/B$

  $B$ misses on every access $\implies Q(n) = \Theta(n^3)$
- Case 2: $c'M^{1/2} < n < cM/B$

  $B$ exploits spatial locality $\implies Q(n) = \Theta(n^3/B)$
- Case 3: $n < cM^{1/2}$

  $B$ fits in cache $\implies Q(n) = \Theta(n^2/B)$

# Blocked (tiled) matrix multiplication

- Introduce a block size $s$:

```
for i_1 = 0, s, ..., n - s                                   //zero-based indexing
  for j_1 = 0, s, ..., n - s
    for k_1 = 0, s, ..., n - s
      for i = i_1, ..., i_1 + s - 1
        for j = j_1, ..., j_1 + s - 1
          for k = k_1, ..., k_1 + s - 1
            C[i * n + j] := C[i * n + j] + A[i * n + k] * B[k * n + j]
```

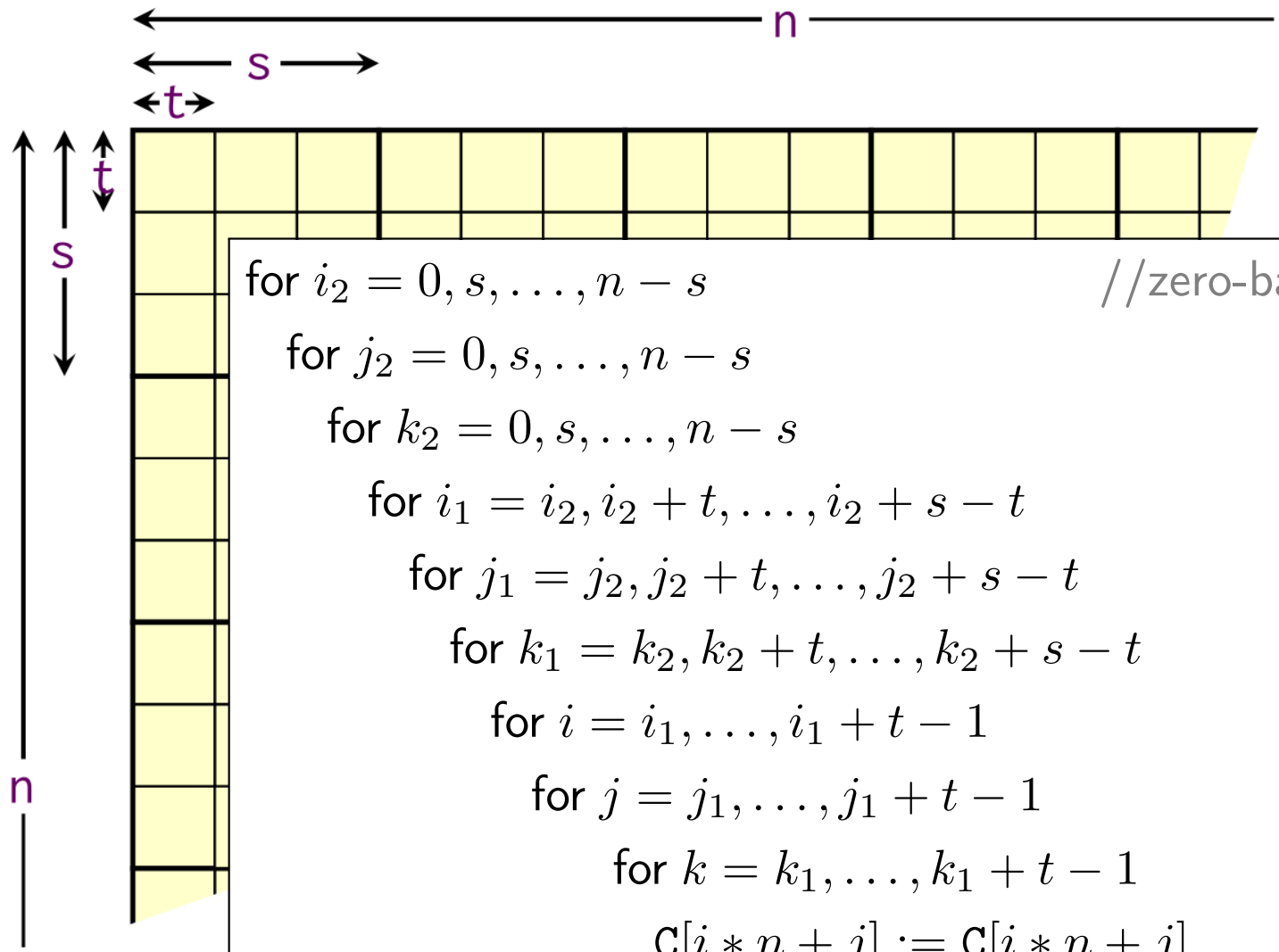
**Tune** $s$ so that the submatrices fit in cache, i.e., $s = \Theta(M^{1/2})$.

$\implies$ **cache-aware** algorithm, **architecture dependent**.

Then $\Theta(s^2/B)$ misses per submatrix so that

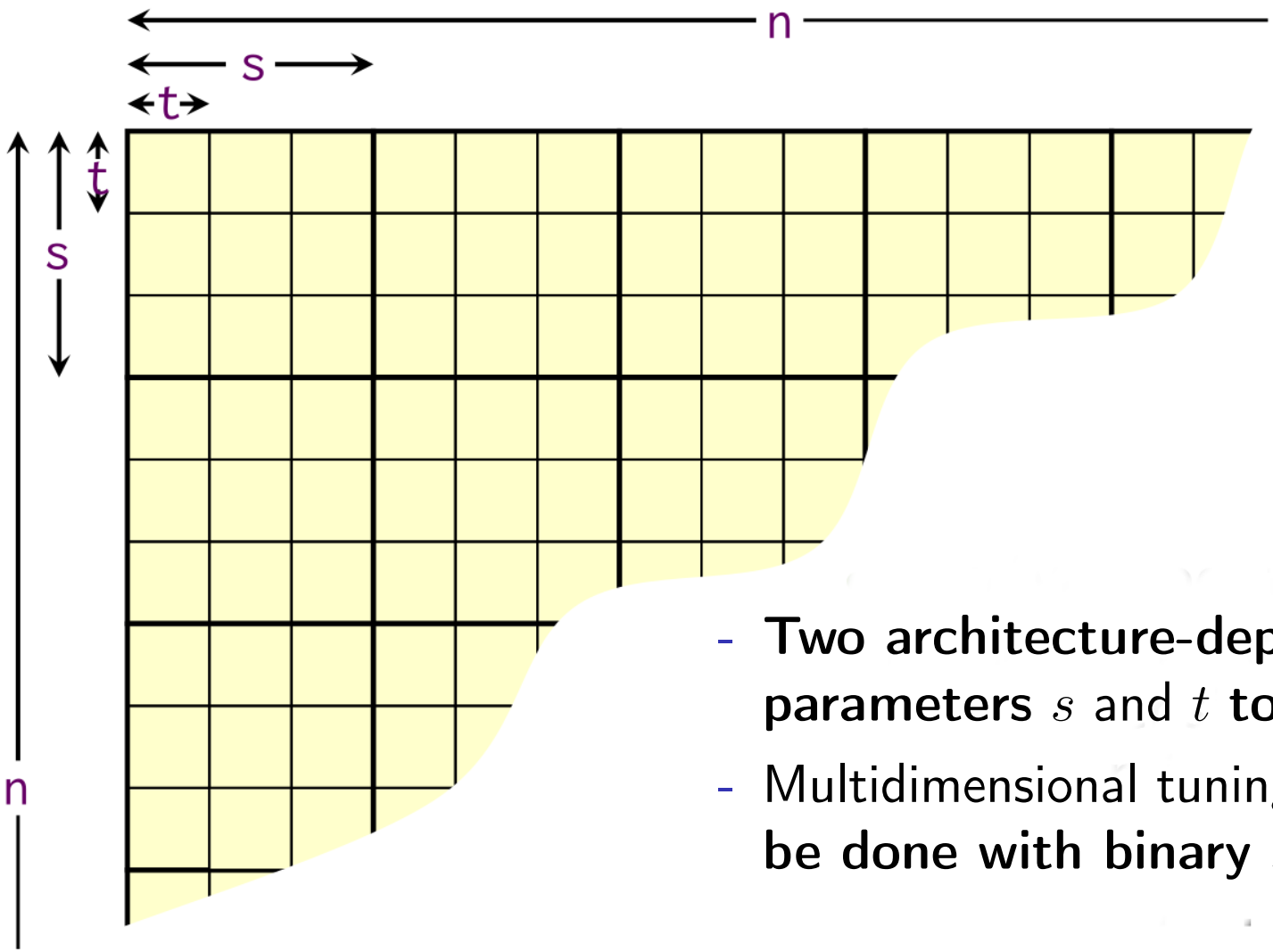
$$Q(n) = \Theta((n/s)^3(s^2/B)) = \Theta(n^3/(BM^{1/2})) \rightarrow \text{ provably } \textbf{optimal}.$$

# Two-level cache



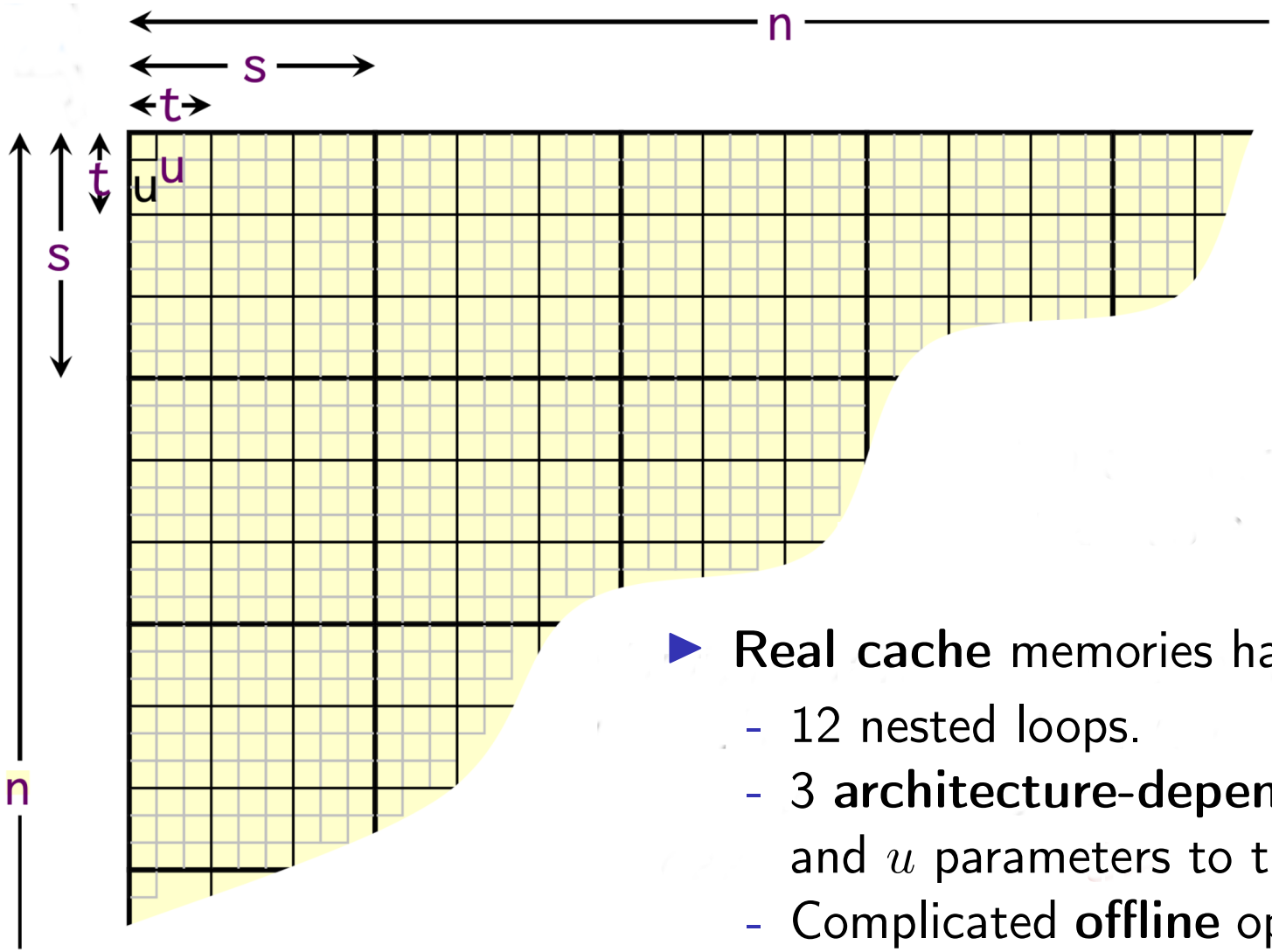

```
for i2 = 0, s, ..., n − s                                   //zero-based indexing
  for j2 = 0, s, ..., n − s
    for k2 = 0, s, ..., n − s
      for i1 = i2, i2 + t, ..., i2 + s − t
        for j1 = j2, j2 + t, ..., j2 + s − t
          for k1 = k2, k2 + t, ..., k2 + s − t
            for i = i1, ..., i1 + t − 1
              for j = j1, ..., j1 + t − 1
                for k = k1, ..., k1 + t − 1
                  C[i ∗ n + j] := C[i ∗ n + j]
                                  + A[i ∗ n + k] ∗ B[k ∗ n + j]
```

# Two-level cache, cont'd




- **Two architecture-dependent parameters $s$ and $t$ to tune.**
- Multidimensional tuning **cannot be done with binary search.**

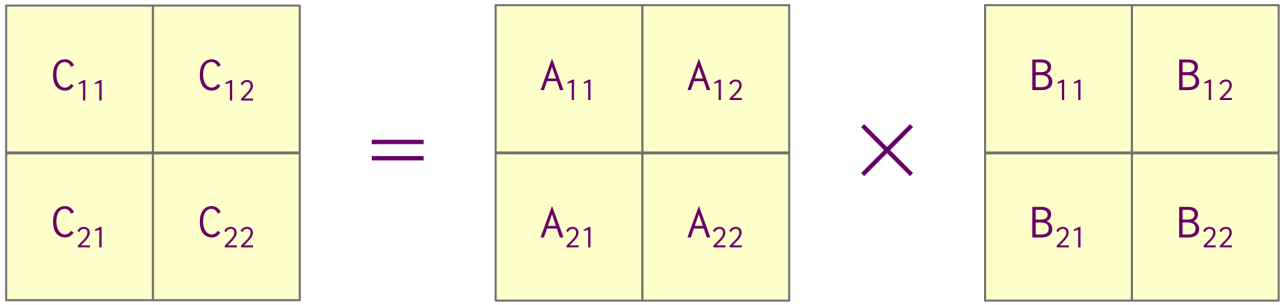
# Three-level cache

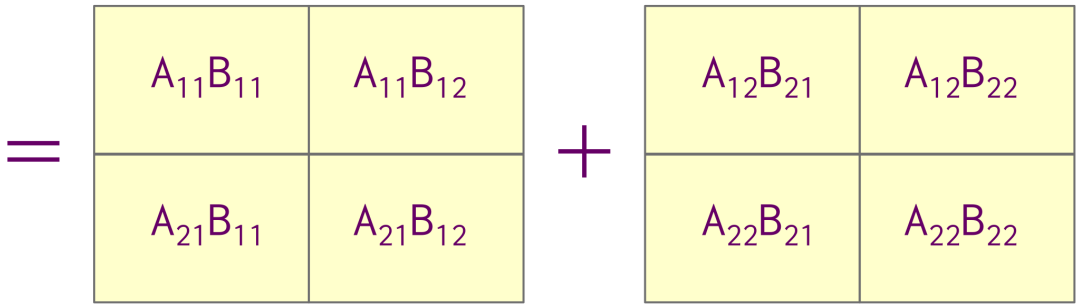



- **Real cache** memories have 3 levels:
  - 12 nested loops.
  - 3 **architecture-dependent** $s$, $t$ and $u$ parameters to tune.
  - Complicated **offline** optimization.
- Existing implementations:
  - OpenBLAS, BLIS, ATLAS.

# Recursive matrix multiplication

- Divide-and-conquer on $n \times n$ matrices:




Recurrence of 8 multiply-and-add of $n/2 \times n/2$ matrices.

There are two cases of memory access:

- Case 1: $n < cM^{1/2} \implies Q(n) = \Theta(n^2/B)$
- Case 2: otherwise $\implies Q(n) = 8Q(n/2) + \Theta(1)$

  (Master theorem) $\implies Q(n) = \Theta(n^3/(BM^{1/2}))$

# Recursive implementation

- Recursive implementation of dense square matrix-matrix multiplication ($\texttt{C} = \texttt{C} + \texttt{AB}$) assuming $n$ is a power of 2:

```
recursive_gemm(A, B, C, n) :                    //C_{1:n,1:n} = 0
  if (n = 1) :
    c_11 := c_11 + a_11 * b_11                  //one-based indexing
  else :
    recursive_gemm(A_{1:n/2,1:n/2}, B_{1:n/2,1:n/2}, C_{1:n/2,1:n/2}, n/2)
    recursive_gemm(A_{1:n/2,n/2+1:n}, B_{n/2+1:n,1:n/2}, C_{1:n/2,n/2+1:n}, n/2)
    recursive_gemm(A_{1:n/2,1:n/2}, B_{1:n/2,n/2+1:n}, C_{1:n/2,n/2+1:n}, n/2)
    recursive_gemm(A_{1:n/2,n/2+1:n}, B_{n/2+1:n,n/2+1:n}, C_{1:n/2,n/2+1:n}, n/2)
    recursive_gemm(A_{n/2+1:n,1:n/2}, B_{1:n/2,1:n/2}, C_{n/2+1:n,1:n/2}, n/2)
    recursive_gemm(A_{n/2+1:n,n/2+1:n}, B_{n/2+1:n,1:n/2}, C_{n/2+1:n,1:n/2}, n/2)
    recursive_gemm(A_{n/2+1:n,1:n/2}, B_{1:n/2:n,n/2+1:n}, C_{n/2+1:n,n/2+1:n}, n/2)
    recursive_gemm(A_{n/2+1:n,n/2+1:n}, B_{n/2+1:n,n/2+1:n}, C_{n/2+1:n,n/2+1:n}, n/2)
```

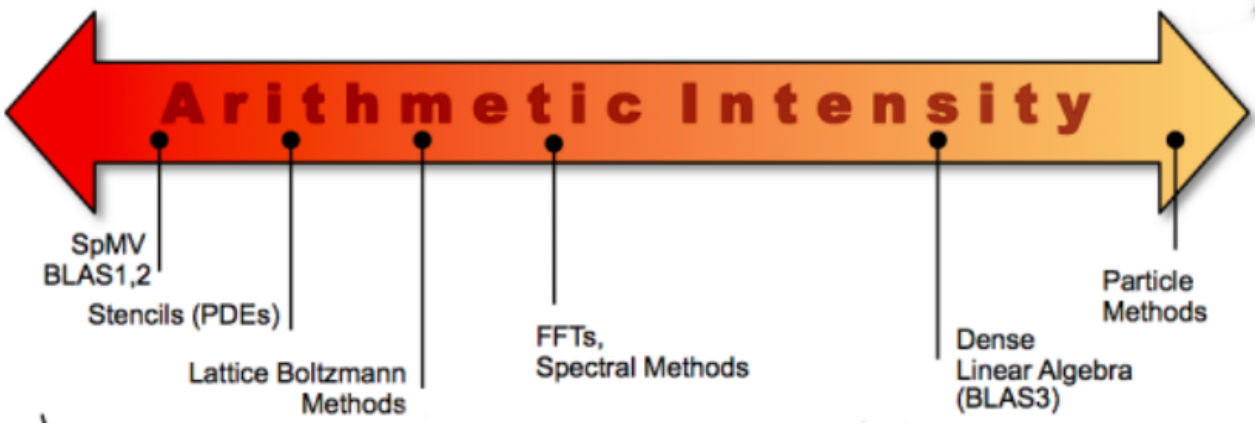
# Efficient cache-oblivious algorithms

- **Cache complexity** of `recursive_gemm` **similar to** that of **tiled matrix multiply**.
- Implementation of `recursive_gemm` **not dependent on memory layout**. It is a **cache-oblivious** algorithm.
- The best cache-oblivious algorithm work on arbitrary rectangular matrices and perform binary splitting (instead of 8-way).
- Recursive matrix-multiply is **typically not implemented in BLAS**, **despite good theoretical cache behavior**, because:
  - It incurs overhead from recursion and function calls.
  - Difficult to efficiently vectorize and parallelize on modern hardware.
  - Hard to outperform hand-optimized, architecture-aware, cache-aware implementations.
- Recursive matrix multiply is more used as a pedagogical tool for illustrating cache efficiency and divide-and-conquer design.
- Recursion is the basis of other fast matrix multiply algorithms, e.g., Strassen's method.

# Overview and principles of communication avoidance

# Reducing data movement

- Principles to reduce communication:
  - **Maximize arithmetic intensity**: Perform more floating-point operations per byte of data moved.




  - **Exploit data locality**: Reuse data in fast memory (cache/registers) as much as possible.
  - **Aggregate communication**: Communicate in bulk instead of fine-grained messages.
- Associated challenges:
  - **Numerical stability**: Communication-avoiding variants may be less stable or harder to analyze.
  - **Architecture-specific tuning**: Optimal strategies depend on cache hierarchy, network topology, etc...

# Sample speedups

- Demmel and Grigori (2016) documents the following speedups achieved by reformulating NLA and other kernels to reduce communication:
  - Up to **12X** faster for **2.5D matrix multiply** on 64K core IBM BG/P
  - Up to **3X** faster for **tensor contractions** on 2K core Cray XE/6
  - Up to **6.2X** faster for **All-Pairs-Shortest-Path** on 24K core Cray CE6
  - Up to **2.1X** faster for **2.5D LU** on 64K core IBM BG/P
  - Up to **11.8X** faster for **direct N-body** on 32K core IBM BG/P
  - Up to **13X** faster for **Tall Skinny QR** on Tesla C2050 Fermi NVIDIA GPU
  - Up to **6.7X** faster for **symeig(band A)** on 10 core Intel Westmere
  - Up to **2X** faster for **2.5D Strassen** on 38K core Cray XT4
  - Up to **4.2X** faster for MiniGMG benchmark bottom solver, using **CA-BiCGStab** (**2.5X** for overall solve)
  - Up to **2.5X** for combustion simulation code

Demmel J. & Grigori L. (2016) Introduction to communication-avoiding algorithms. CS294/Math270, UC Berkeley.

# Sparse matrix-vector product (SpMV)

# Low data locality of SpMV

- Recall the CSR data structure:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & a_{43} & 0 \end{bmatrix}$$

$$\texttt{val} = [a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{33}, a_{34}, a_{43}]$$

$$\texttt{col\_idx} = [1, 2, 3, 1, 2, 3, 4, 3]$$

$$\texttt{row\_start} = [1, 4, 6, 8, 9]$$

with the following SpMV kernel ($\texttt{y} := \texttt{y} + \texttt{Ax}$):

```
for i = 1, ..., n                                   //one-based indexing
  for j = row_start[i], ..., row_start[i + 1] - 1
    y[i] := y[i] + val[j] * x[col_idx[j]]
```

# Low data locality of SpMV, cont'd

- SpMV kernel ($\mathtt{y} := \mathtt{y} + \mathtt{Ax}$) for CSR data structures:

$$\begin{array}{l}\text{for } i = 1, \ldots, n \qquad\qquad \text{//one-based indexing} \\ \quad \text{for } j = \mathtt{row\_start}[i], \ldots, \mathtt{row\_start}[i+1] - 1 \\ \quad\quad \mathtt{y}[i] := \mathtt{y}[i] + \mathtt{val}[j] * \mathtt{x}[\mathtt{col\_idx}[j]]\end{array}$$

- **Irregular access** to the components of $\mathtt{x}$:
  - **No spatial locality**, **no temporal locality**.
  - **Every component** of $\mathtt{x}$ loaded for a single multiply-and-add is **trashed immediatly from register**. **No loop unrolling**, **no SIMD**, ...
  - **Sparsity induces lower arithmetic intensity** (AI) than for general dense matrix-vector product (GEMV) kernels.
  - **SpMV kernels** are **memory-bound**, even if $\mathtt{x}$ fits in cache.
  - **Performance** of SpMV kernels depends on **data structure**, **non-zero structure** and **hardware**.

# Register blocking

- Improve data locality by promoting **register reuse**:
  - **Register reuse** is **achieved** for the source vector **by blocking**, also known as **tiling**, i.e., relying on contiguous storage of non-zero blocks in the matrix.
  - **Registers** are small, i.e., only **store a handfull of data**:

    $\Longrightarrow$ small $r \times c$ block sizes : $r$ values of destination vector + $c$ values of source vector + $rc$ values of a block are stored in register.
  - Blocking allows for **compiler optimizations**, e.g., loop unrolling, pipelining, ...
  - **Optimal block size dependent** on **sparsity strucutre** and **machine architecture**.
  - Considerable **overhead** in changing data structure for register blocking.
  - Applications such as finite element methods (FEM) naturally store small dense blocks in sparse matrices.
  - Most matrices do not have uniform block structures:

    $\Longrightarrow$ **some zero values** are **stored** for non-fully dense blocks.

# Blocked SpMV

- Recall the BSR data structure format:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & a_{43} & 0 \end{bmatrix}$$

$$\texttt{r} = 2,\ \texttt{c} = 2$$

$$\texttt{val} = [a_{11}, a_{12}, a_{21}, a_{22}, a_{13}, 0, 0, 0, a_{33}, a_{34}, a_{43}, 0]$$

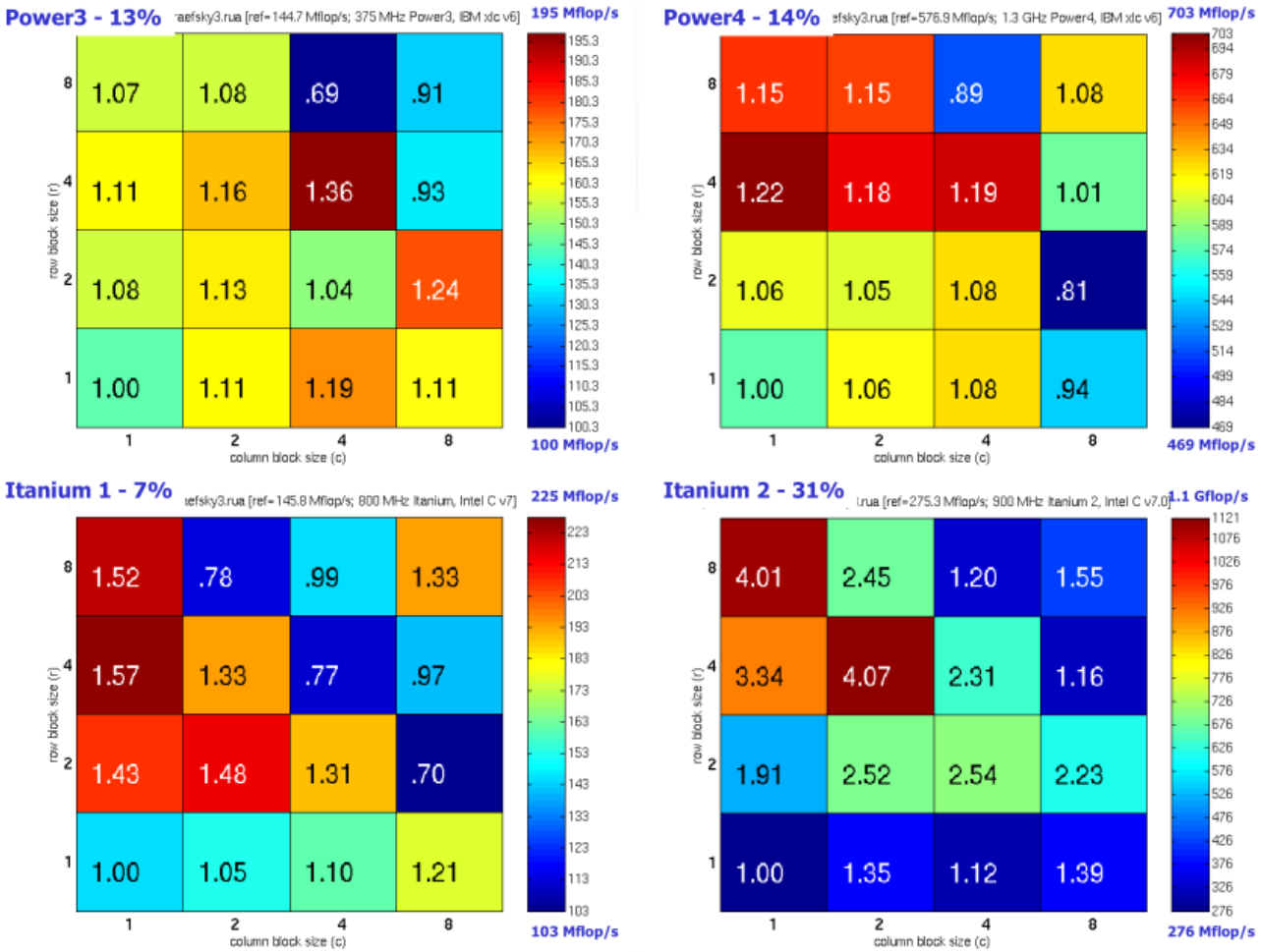
$$\texttt{col\_idx} = [1, 2, 2],\ \texttt{row\_start} = [1, 3, 4]$$

with the following SpMV kernel ($\texttt{y} := \texttt{y} + \texttt{Ax}$):

for $i = 1, \ldots, n/r$ //one-based indexing
  for $j = \texttt{row\_start}[i], \ldots, \texttt{row\_start}[i+1] - 1$
    for $ii = 1, \ldots, r$
      for $jj = 1, \ldots, c$
        $\texttt{y}[(i-1) * r + ii] := \texttt{y}[(i-1) * r + ii] +$
        $\texttt{val}[(j-1) * rc + (ii-1) * c + jj] * \texttt{x}[(\texttt{col\_idx}[j] - 1) * rc + jj]$

# Tuning of register blocks

- Optimal block size for register reuse depends on **sparsity** and **machine architecture**:
  - Automated tuning interface by Vuduc et al. (2005): OSKI




Vuduc, R., Demmel, J. W., & Yelick, K. A. (2005). OSKI: A library of automatically tuned sparse matrix kernels. In Journal of Physics: Conference Series (Vol. 16, No. 1, p. 521). IOP Publishing.

# Encoding block sparsity

- **Zeros stored** in non-fully dense blocks **result in** extra floating-point operations, and more importantly, **extra data movement**.
- Only non-zero values need be stored and fetched when **local block sparsity** is **encoded**:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & a_{43} & 0 \end{bmatrix}$$

$$\texttt{r} = 2, \texttt{c} = 2$$

$$\texttt{val} = [a_{11}, a_{12}, a_{21}, a_{22}, a_{13}, a_{33}, a_{34}, a_{43}]$$

$$\texttt{col\_idx} = [1, 2, 2],\ \texttt{row\_start} = [1, 3, 4]$$

$$\texttt{b\_map} = [15, 1, 7]$$

E.g., $\begin{bmatrix} a_{33} & a_{34} \\ a_{43} & 0 \end{bmatrix} \implies$ sparsity pattern $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \implies$ unrolled to $1110$

$$\implies 1 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 = 7.$$

- **if statements** in SpMV's loop **decrease efficiency** $\implies$ two approaches: operate on zeros (Buluç et al., 2011), de Bruijn sequences (Kannan, 2013).

Buluç, A., Williams, S., Oliker, L., & Demmel, J. (2011). Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In 2011 IEEE International Parallel & Distributed Processing Symposium (pp. 721-733)

Kannan, R. (2013). Efficient sparse matrix multiple-vector multiplication using a bitmapped format. In 20th Annual International Conference on High Performance Computing (pp. 286-294).

# Block Gram-Schmidt procedures

# Block Gram-Schmidt procedures

- **Gram-Schmidt (GS)** procedure:
  - Returns the QR factorization of an $m \times n$ matrix $X$ by **orthogonalizing vectors**, **one-at-a-time**, against previously orthogonalized vectors.
- **Block Gram-Schmidt (BGS)** procedure:
  - Works on **blocks of $s$ vectors** at a time.
  - Procedure fully specified upon definition of a **projector $\Pi^{(i)}$ onto** range$(Q_{:,1:is})^{\perp}$ and an **IntraOrtho** orthogonalization procedure:

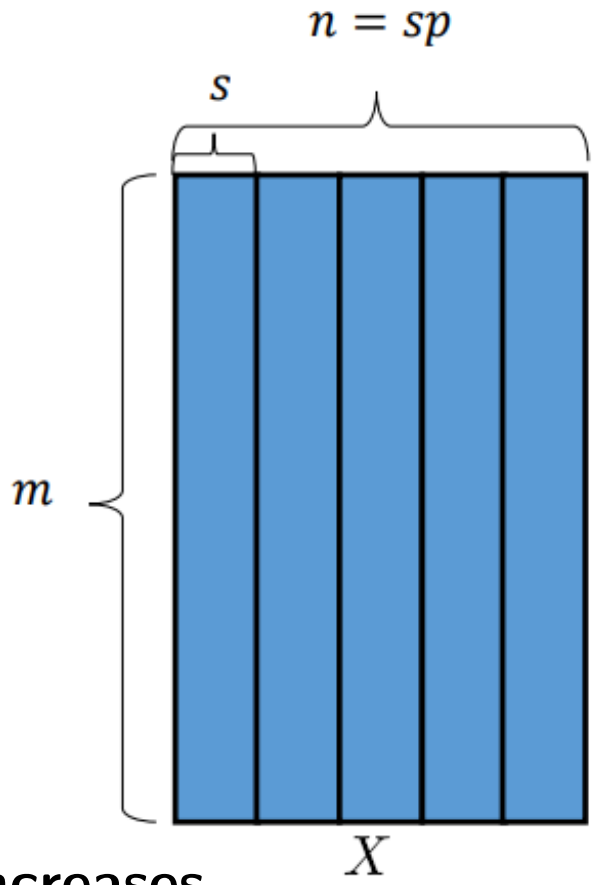
IntraOrtho$(X_{:,1:s}) \mapsto (Q_{:,1:s}, R_{1:s,1:s})$

for $i = 2, \ldots, p$ //one-based indexing

$Q_{:,(i-1)s+1:is} := \Pi^{(i-1)} X_{:,(i-1)s+1:is}$

IntraOrtho$(Q_{:,(i-1)s+1:is}) \mapsto$

$(Q_{:,(i-1)s+1:is}, R_{(i-1)s+1:is,(i-1)s+1:is})$




  - Applying $\Pi^{(i)}$ to blocks of $s$ vectors at a time **increases arithmetic intensity**, and **decreases synchronizations**.

# Block classical Gram-Schmidt (BCGS)

- BCGS is formed with $\Pi^{(i)} := I_n - Q_{:,1:is}Q_{:,1:is}^T$ which leads to

IntraOrtho$(X_{:,1:s}) \mapsto (Q_{:,1:s}, R_{1:s,1:s})$ //one-based indexing

for $i = 2, \ldots, p$

$R_{1:(i-1)s,(i-1)s+1:is} := Q_{:,1:(i-1)s}^T X_{:,(i-1)s+1:is}$ //BLAS-3

$W := X_{:,(i-1)s+1:is} - Q_{:,1:(i-1)s}R_{1:(i-1)s,(i-1)s+1:is}$ //BLAS-3

IntraOrtho$(W) \mapsto (Q_{:,(i-1)s+1:is}, R_{(i-1)s+1:is,(i-1)s+1:is})$

**All $s$ vectors** of the block treated by IntraOrtho are **simultaneously available**.

Therefore, **IntraOrtho needs not** necessarily **be Gram-Schmidt**.

The QR factorization can be computed by Cholesky QR (CholQR), tall and skinny QR (TSQR), Householder QR (HouseQR), ...:

IntraOrtho $\in \{$CGS, CGS2, MGS, CholQR, CholQR2, TSQR, HouseQR, $\ldots\}$

Each so-specified variant is referred to as BCSG $\circ$ IntraOrtho.

# Block classical Gram-Schmidt reorthogonalized (BCGS2)

- BCGS is stabilized by reorthogonalizing either by running BCSG twice, or by repeating each inner loop (BCGS2):

IntraOrtho$(X_{:,1:s}) \mapsto (Q_{:,1:s}, R_{1:s,1:s})$ //one-based indexing

for $i = 2, \ldots, p$

$R^{(1)}_{1:(i-1)s,(i-1)s+1:is} := Q^T_{:,1:(i-1)s} X_{:,(i-1)s+1:is}$ //BLAS-3

$W := X_{:,(i-1)s+1:is} - Q_{:,1:(i-1)s} R^{(1)}_{1:(i-1)s,(i-1)s+1:is}$ //BLAS-3

IntraOrtho$(W) \mapsto (\widehat{Q}, R^{(1)}_{(i-1)s+1:is,(i-1)s+1:is})$

$R^{(2)}_{1:(i-1)s,(i-1)s+1:is} := Q^T_{:,1:(i-1)s} \widehat{Q}$ //BLAS-3

$W := \widehat{Q} - Q_{:,1:(i-1)s} R^{(2)}_{1:(i-1)s,(i-1)s+1:is}$ //BLAS-3

IntraOrtho$(W) \mapsto (Q_{:,(i-1)s+1:is}, R^{(2)}_{(i-1)s+1:is,(i-1)s+1:is})$

$R_{1:(i-1)s,(i-1)s+1:is} := R^{(1)}_{1:(i-1)s,(i-1)s+1:is} + R^{(2)}_{1:(i-1)s,(i-1)s+1:is} R^{(1)}_{(i-1)s+1:is,(i-1)s+1:is}$

$R_{(i-1)s+1:is,(i-1)s+1:is} = R^{(2)}_{(i-1)s+1:is,(i-1)s+1:is} R^{(1)}_{(i-1)s+1:is,(i-1)s+1:is}$

We refer to associated variants as BCGS2 $\circ$ IntraOrtho where IntraOrtho $\in \{$CGS, CGS2, MGS, CholQR, CholQR2, TSQR, HouseQR, $\ldots\}$

# Block modified Gram-Schmidt (BMGS)

- BMGS is formed with

$$\Pi^{(i)} := (I_n - Q_{:,(i-1)s+1:is} Q_{:,(i-1)s+1:is}^T) \cdots (I_n - Q_{:,1:s} Q_{:,1:s}^T)$$

which leads to

IntraOrtho$(X_{:,1:s}) \mapsto (Q_{:,1:s}, R_{1:s,1:s})$ //one-based indexing

for $i = 2, \ldots, p$

$W := X_{:,(i-1)s+1:is}$

for $j = 1, \ldots, i-1$

$R_{(j-1)s+1:js,(i-1)s+1:is} := Q_{:,(j-1)s+1:js}^T W$ //BLAS-3

$W := W - Q_{:,(j-1)s+1:js} R_{(j-1)s+1:js,(i-1)s+1:is}$

IntraOrtho$(W) \mapsto (Q_{:,(i-1)s+1:is}, R_{(i-1)s+1:is,(i-1)s+1:is})$

We refer to associated variants as BMGS $\circ$ IntraOrtho where

IntraOrtho $\in \{$CGS, CGS2, MGS, CholQR, CholQR2, TSQR, HouseQR, $\ldots\}$

# Block randomized Gram-Schmidt (BRGS)

- BRGS is formed by with $\Pi^{(i)} := I_n - Q_{:,1:is}(\Theta Q_{:,1:is})^\dagger \Theta$ which leads to

Compute $Q_{:,1:s}$ and $R_{1:s,1:s}$ //RGS or RCholQR

s.t. $\Theta Q_{:,1:s} R_{1:s,1:s} = \Theta X_{:,1:s}$

and $(\Theta Q_{:,1:s})^T \Theta Q_{:,1:s} = I_s$

$S_{:,1:s} := \Theta Q_{:,1:s}$ //one-based indexing

for $i = 2, \ldots, p$

$P := \Theta X_{:,(i-1)s+1:is}$

$R_{1:(i-1)s,(i-1)s+1:is} := \arg \min_{Y \in \mathbb{R}^{(i-1)s \times s}} \|S_{:,1:(i-1)s} Y - P\|_F$

$Q_{:,(i-1)s+1:is} := X_{:,(i-1)s+1:is} - Q_{:,1:(i-1)s} R_{1:(i-1)s,(i-1)s+1:is}$

Compute $Q_{:,(i-1)s+1:is}$ and $R_{(i-1)s+1:is,(i-1)s+1:is}$ //RGS or RCholQR

s.t. $\Theta Q_{:,(i-1)s+1:is} R_{(i-1)s+1:is,(i-1)s+1:is} = \Theta X_{:,(i-1)s+1:is}$

and $(\Theta Q_{:,(i-1)s+1:is})^T \Theta Q_{:,(i-1)s+1:is} = I_s$

$S_{:,(i-1)s+1:is} := \Theta Q_{:,(i-1)s+1:is}$

# Stability aspects

- The loss of orthogonality $\|I_n - Q^T Q\|_2$ achieved in finite precision by BGS algorithms depends on $\Pi^{(i)}$ as well as on IntraOrtho.

  If IntraOrtho is unconditionally stable, i.e., if IntraOrtho$(X) \mapsto (Q, R)$ is such that $\|I_n - Q^T Q\|_2 = \mathcal{O}(\varepsilon)$ irrespective of $\kappa(X)$, then:

| BGS | $\|I_n - Q^T Q\|_2$ | Assumption on $\kappa(X)$ | Reference(s) |
|---|---|---|---|
| BCGS | $\mathcal{O}(\varepsilon)\kappa^{n-1}(X)$ | $\mathcal{O}(\varepsilon)\kappa(X) < 1$ | conjecture |
| BCGS2 | $\mathcal{O}(\varepsilon)$ | $\mathcal{O}(\varepsilon)\kappa(X) < 1$ | Barlow and Smoktunowicz (2013) |
| BMGS | $\mathcal{O}(\varepsilon)\kappa(X)$ | $\mathcal{O}(\varepsilon)\kappa(X) < 1$ | Jalby and Philippe (1991) |

- Most orthogonalization procedures are not unconditionally stable:

| IntraOrtho | $\|I_n - Q^T Q\|_2$ | Assumption on $\kappa(X)$ | Reference(s) |
|---|---|---|---|
| CGS | $\mathcal{O}(\varepsilon)\kappa^{n-1}(X)$ | $\mathcal{O}(\varepsilon)\kappa(X) < 1$ | Kiełbasiński (1974) |
| CGS2 | $\mathcal{O}(\varepsilon)$ | $\mathcal{O}(\varepsilon)\kappa(X) < 1$ | Giraud et al. (2005) |
| MGS | $\mathcal{O}(\varepsilon)\kappa(X)$ | $\mathcal{O}(\varepsilon)\kappa(X) < 1$ | Björck (1967) |
| CholQR | $\mathcal{O}(\varepsilon)\kappa^2(X)$ | $\mathcal{O}(\varepsilon)\kappa^2(X) < 1$ | Yamamoto et al. (2015) |
| CholQR2 | $\mathcal{O}(\varepsilon)$ | $\mathcal{O}(\varepsilon)\kappa^2(X) < 1$ | Yamamoto et al. (2015) |
| HouseQR | $\mathcal{O}(\varepsilon)$ | none | Section 10 in Higham (2002) |
| TSQR | $\mathcal{O}(\varepsilon)$ | none | Mori et al. (2012) |

# Stability aspects, cont'd

- Other results available:

  Jalby and Philippe (1991):
  - BMGS $\circ$ MGS behaves like CGS.
  - BMGS $\circ$ MGS2 is as stable as MGS.

  Carson et al. (2022):
  - BMGS $\circ$ any IntraOrtho with $\|I_n - Q^T Q\| \in \mathcal{O}(\varepsilon)$ is as stable MGS.


Balabanov, O., & Grigori, L. (2025). Randomized block Gram–Schmidt process for the solution of linear systems and eigenvalue problems. SIAM Journal on Scientific Computing, 47(1), A553-A585.

Barlow, J. L., & Smoktunowicz, A. (2013). Reorthogonalized block classical Gram–Schmidt. Numerische Mathematik, 123(3), 395-423.

Björck, Å. (1967). Solving linear least squares problems by Gram-Schmidt orthogonalization. BIT Numerical Mathematics, 7(1), 1-21.

Carson, E., Lund, K., Rozložník, M., & Thomas, S. (2022). Block Gram-Schmidt algorithms and their stability properties. Linear Algebra and its Applications, 638, 150-195.

Giraud, L., Langou, J., Rozložník, M., & Eshof, J. V. D. (2005). Rounding error analysis of the classical Gram-Schmidt orthogonalization process. Numerische Mathematik, 101(1), 87-100.

Higham, N. J. (2002). Accuracy and stability of numerical algorithms. Society for industrial and applied mathematics.

Jalby, W., & Philippe, B. (1991). Stability analysis and improvement of the block Gram–Schmidt algorithm. SIAM journal on scientific and statistical computing, 12(5), 1058-1073.

Kiełbasiński, A. (1974). Analiza numeryczna algorytmu ortogonalizacji Grama-Schmidta. Mathematica Applicanda, 2(2).

Mori, D., Yamamoto, Y., & Zhang, S. L. (2012). Backward error analysis of the AllReduce algorithm for Householder QR decomposition. Japan journal of industrial and applied mathematics, 29, 111-130.

Yamamoto, Y., Nakatsukasa, Y., Yanagisawa, Y., & Fukaya, T. (2015). Roundoff error analysis of the CholeskyQR2 algorithm. Electron. Trans. Numer. Anal, 44(01), 306-326.

# $s$-step iterative solvers

# Introduction to $s$-step Krylov subspace methods

- Krylov subspace methods seek for iterates in some Krylov subspace $\mathcal{K}_m$ with residual orthogonal to some subspace $\mathcal{L}_m$.
  At each iteration, the following steps are executed:
  1. Increase the dimension of $\mathcal{K}_m$:
     Sparse matrix-vector (**SpMV**) product (**communication-bound**)
  2. Orthogonalize against $\mathcal{L}_m$:
     **Gram-Schmidt** procedure (**communication-bound**)

  $\Longrightarrow$ **Performance** of Krylov subspace methods **limited by comunication**.
- $s$-step Krylov subspace methods **compute blocks of $s$ iterations** at once:
  - Iteration loop split into an **outer loop** and an **inner loop**.

    SpMV $(Av) \rightarrow$ **Matrix power kernel** $(Av, \ldots, A^s v)$

    Gram-Schmidt $\rightarrow$ **Block Gram-Schmidt**
  - Mathematically equivalent to standard methods (in perfect arithmetic).

  $\Longrightarrow$ **Increases arithmetic intensity** (sequential implementation),
  **Reduces number of synchronizations** (parallel implementation).

# s-step Arnoldi

- An $s$-step Arnoldi procedure computes a basis $\mathfrak{Q}_j := [Q_1, \ldots, Q_j]$ of the Krylov subspace range$([V_1, \ldots, V_j]) = \mathcal{K}_{js}(A, v)$ using a projector $\Pi^{(j)}$ onto range$(\mathfrak{Q}_j)^\perp$ and an IntraOrtho procedure:

$V_1 := [v, Av, \ldots, A^{s-1}v]$
IntraOrtho$(V_1) \mapsto (Q_1, R_1)$
for $j = 2, \ldots, p$
  $v := V_{j-1}[:, s]$ //one-based indexing
  $V_j := [Av, \ldots, A^s v]$ //matrix power kernel
  $Q_j := \Pi^{(j-1)} V_j$
  IntraOrtho$(Q_j) \mapsto (Q_j, R_j)$

  - Hoemmen (2010) uses $\Pi^{(j)} := I_n - \mathfrak{Q}_j \mathfrak{Q}_j^T$ (BCGS), IntraOrtho =TSQR.
  - In parallel, $s$-step Arnoldi requires $s$X less messages than regular Arnoldi.
  - Loss of orthogonality increases rapidly with $s$, delaying convergence $\Longrightarrow$ optimal $s$ depends on $A$ and architecture.

Hoemmen, M. (2010). Communication-avoiding Krylov subspace methods. PhD thesis. University of California, Berkeley.

# s-step Arnoldi, cont'd

- In practice, the ill-conditioning grows so quick that $s$ values need to be kept small, i.e., typically no more than 5.

  Two ways to improve stability:

  - Use a more stable BGS kernel than BCGS, e.g., BMGS or BCGS2.
  - Use Chebyshev or Newton matrix polynomials instead of monomials.

  Arbitrary matrix polynomials are used as follows:

$$
\begin{array}{l}
V_1 := [p_0(A)v, p_1(A)v, \ldots, p_{s-1}(A)v] \\
\text{IntraOrtho}(V_1) \mapsto (Q_1, R_1) \\
\text{for } j = 2, \ldots, p \\
\quad v := V_{j-1}[:, s] \qquad\qquad \text{//one-based indexing} \\
\quad V_j := [p_1(A)v, \ldots, p_s(A)v] \quad \text{//matrix power kernel} \\
\quad Q_j := \Pi^{(j-1)} V_j \\
\quad \text{IntraOrtho}(Q_j) \mapsto (Q_j, R_j)
\end{array}
$$

  - $s$-step Arnoldi forms the basis for $s$-tep GMRES.

# Matrix power kernels

# Matrix power kernels

- An essential part of $s$-step methods is the **computation** of $s$ **matrix powers**

$$p_1(A)v, \ldots, p_s(A)v$$

which relies on **memory-bound SpMV** calls.

In a **sequential** computation, $s$ SpMV invocations require **reading the entire matrix $s$ times from slow memory**.

In **parallel**, they require $\Omega(s)$ **messages** (if the matrix is not block diagonal).

- Demmel et al. (2007, 2008, 2009) develop **communication-avoiding matrix power kernels**:
  - Requires $1 + o(1)$ read of the sparse matrix $A$ for **sequential** implementations.
  - Requires only $\mathcal{O}(1)$ messages in **parallel**.

Demmel J.W., Hoemmen M., Mohiyuddin M. & Yelick K.A. (2007). Avoiding communication in computing Krylov subspaces. Tech. Rep. UCB/EECS-2007-123, University of California, Berkeley.

Demmel J.W., Hoemmen M., Mohiyuddin M. & Yelick K.A. (2008). Avoiding communication in sparse matrix computations. IEEE International Parallel and Distributed Processing Symposium.

Demmel J.W., Hoemmen M., Mohiyuddin M. & Yelick K.A. (2009). Minimizing communication in sparse matrix solvers. Proceedings of the 2009 ACM/IEEE Conference on Supercomputing (New York, NY, USA).

# Homework problems

# Homework problem

Turn in **your own** solution to **Pb. 35**:

**Pb. 35** Consider a sparse matrix $A$ of size $10^6 \times 10^6$ with $5 \cdot 10^6$ non-zero values. Analyze the worst case memory access pattern in terms of number of read and write accesses and find the corresponding arithmetic intensity of standard CSR-based SpMV for this matrix.

**Pb. 36** Consider a sparse matrix $A$ of size $10^6 \times 10^6$ with $5 \cdot 10^6$ non-zero values spread in dense $4 \times 4$ blocks. Let the matrix be stored in BSR format with a block size of $r \times c = 4 \times 4$. You may further assume that all stored blocks of the BSR data structure are perfectly aligned with the non-zero pattern of the matrix. Assuming that, within blocks, there is a perfect reuse of data from the source and destination vectors in registers, analyze the worst case memory access pattern in terms of number of read and write accesses, and find the corresponding arithmetic intensity of the BSR-based SpMV kernel for this matrix. Ignore indexing arithmetic in your account of floating-point operations.

# Practice session

# Practice session

1. Code a function `BCGS` with the option to use either of CGS, CGS2, MGS, CholQR and CholQR2 as IntraOrtho.
2. Code a function `BCGS2` with the option to use either of CGS, CGS2, MGS, CholQR and CholQR2 as IntraOrtho.
3. Code a function `BMGS` with the option to use either of CGS, CGS2, MGS, CholQR and CholQR2 as IntraOrtho.
4. Code a function `BRGS` with the option to use either of RGS and RCholQR as IntraOrtho.
5. Compute QR factorizations of tall and skinny matrices using both BGS procedures with block sizes $s \in \{5, 10\}$ and GS procedures. Compare running times and plot loss of orthogonality, i.e., $\|I_m - Q^T Q\|_2$.