



Single processor tuning (2/2)

Prof. Richard Vuduc

Georgia Institute of Technology

CSE/CS 8803 PNA: Parallel Numerical Algorithms

[L.16] Thursday, February 28, 2008



Today's sources

- CS 267 (Demmel & Yelick @ UCB; Spring 2007)
- “*A family of high-performance matrix multiplication algorithms,*” by Gunnels, *et al.* (2006)
- “*Anatomy of high-performance matrix multiplication,*” by Goto and van de Geijn (2006)
- “*An experimental comparison of cache-oblivious and cache-conscious programs?*” by Yotov, *et al.* (SPAA 2007)
- Talk by Matteo Frigo at CScADS Autotuning Workshop (2007)



Review: GPGPUs.

(I don't know; you tell me!)



Review:
A one-level model of the
memory hierarchy



A simple model of memory

m \equiv No. words moved from slow to fast memory

f \equiv No. of flops

α \equiv Time per slow memory op.

τ \equiv Time per flop

q \equiv $\frac{f}{m}$ = Flop-to-mop ratio \Leftarrow Computational intensity

$$T = f \cdot \tau + m \cdot \alpha = f \cdot \tau \cdot \left(1 + \frac{\alpha}{\tau} \cdot \frac{1}{q} \right)$$

Machine balance



Blocked (tiled) matrix multiply

// Let $I, J, K =$ blocks of b indices

for $I \leftarrow$ index blocks 1 to $\frac{n}{b}$ do

for $J \leftarrow$ index blocks 1 to $\frac{n}{b}$ do

// Read block C_{IJ}

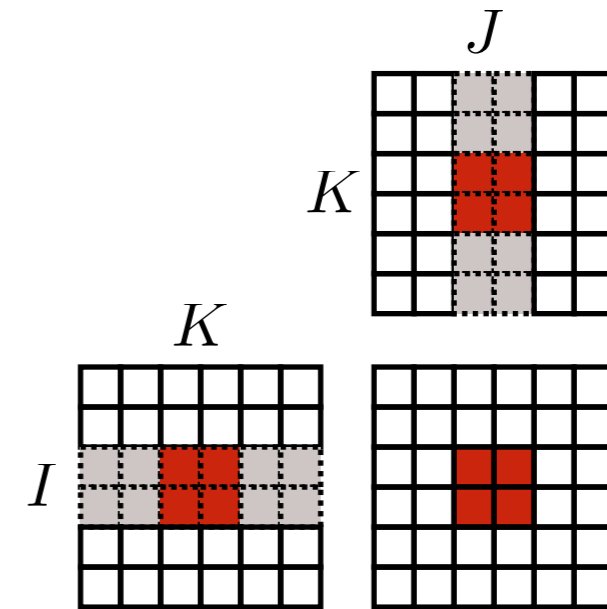
for $K \leftarrow$ index blocks 1 to $\frac{n}{b}$ do

// Read block A_{IK}

// Read block B_{KJ}

$C_{IJ} \leftarrow C_{IJ} + A_{IK} \cdot B_{KJ}$

// Write C_{IJ} to slow memory



$$m \approx \frac{n^3}{b} \implies q \approx b$$
$$\frac{T}{f \cdot \tau} = 1 + \frac{\alpha}{\tau} \cdot \frac{1}{b}$$

Can we do better? Nope.

- **Theorem** [Hong and Kung (1981)]: Any schedule of conventional matrix multiply must transfer $\Omega(n^3 / \sqrt{M})$ words between slow and fast memory, where $M < n^2 / 6$.
- Last time: We did intuitive proof by Toledo (1999)
- Historical note: Rutledge & Rubinstein (1951–52)
- So cached block matrix multiply is **asymptotically optimal**.

$$b = O\left(\sqrt{M}\right) \implies m = O\left(\frac{n^3}{b}\right) = O\left(\frac{n^3}{\sqrt{M}}\right)$$



Architectural implications

Arch.	$\approx \alpha / \tau$	M
Ultra 2i	25	1.5 MB
Ultra 3	14	460 KB
Pentium 3	6.3	94 KB
P-3M	10	240 KB
Power3	8.8	180 KB
Power4	15	527 KB
Itanium 1	36	3.0 MB
Itanium 2	5.5	71 KB

$M \equiv$ Size of fast mem.

$$3b^2 \leq M$$

$$q \approx b$$

\Downarrow

$$M \geq 3q^2$$

$$1 + \frac{\alpha}{\tau} \cdot \frac{1}{q} < 1.1$$

$$\implies M \geq 300 \left(\frac{\alpha}{\tau} \right)^2$$

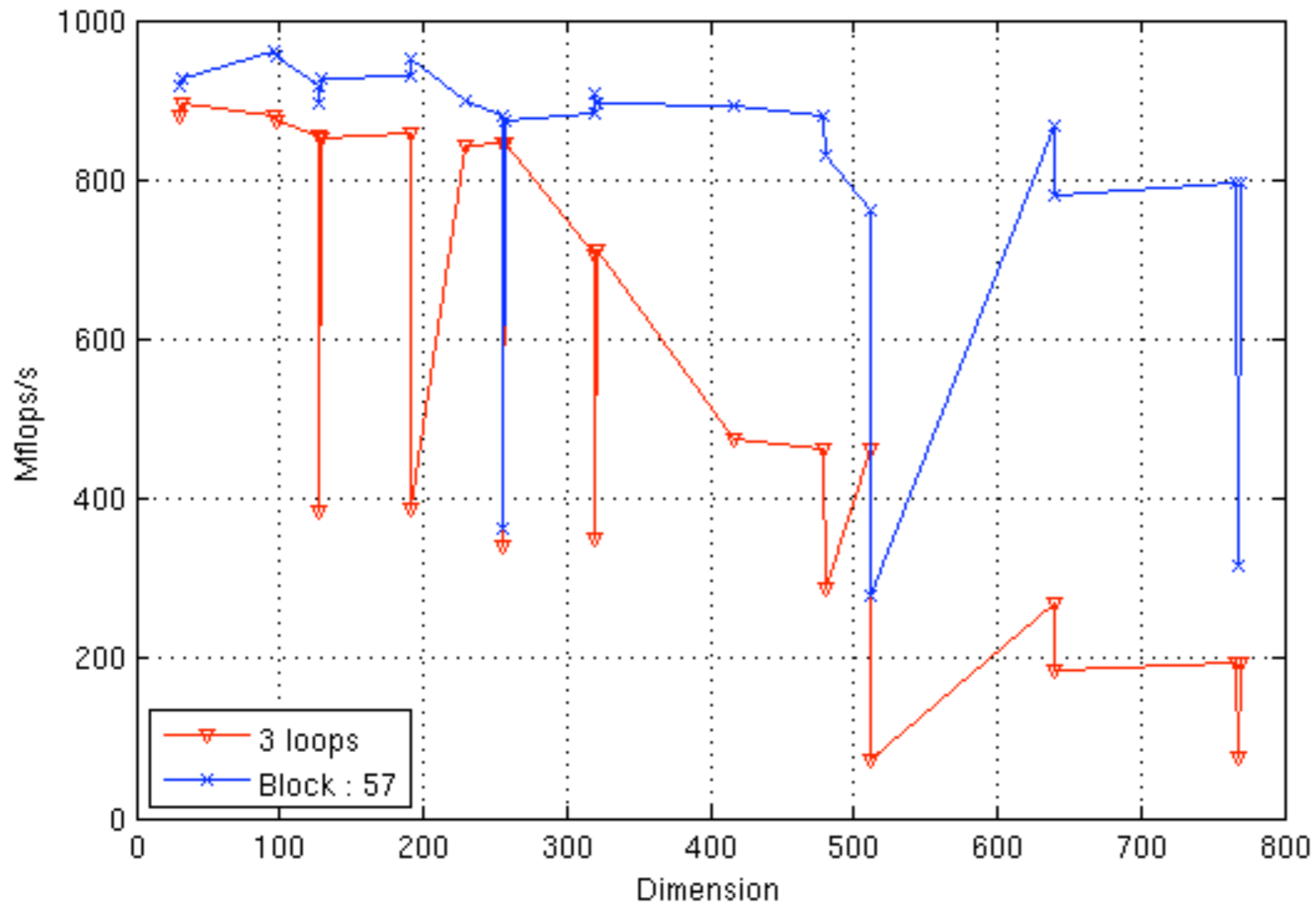
Note: "M" in bytes to 2 digits; assumes 8-byte (double-precision) words



What happens in practice?

- Experiment: One-level cache-blocked matrix multiply
- Block size chosen as square, by exhaustive search over sizes up to 64

Tiled MM on AMD Opteron 2.2 GHz (4.4 Gflop/s peak), 1 MB L2 cache



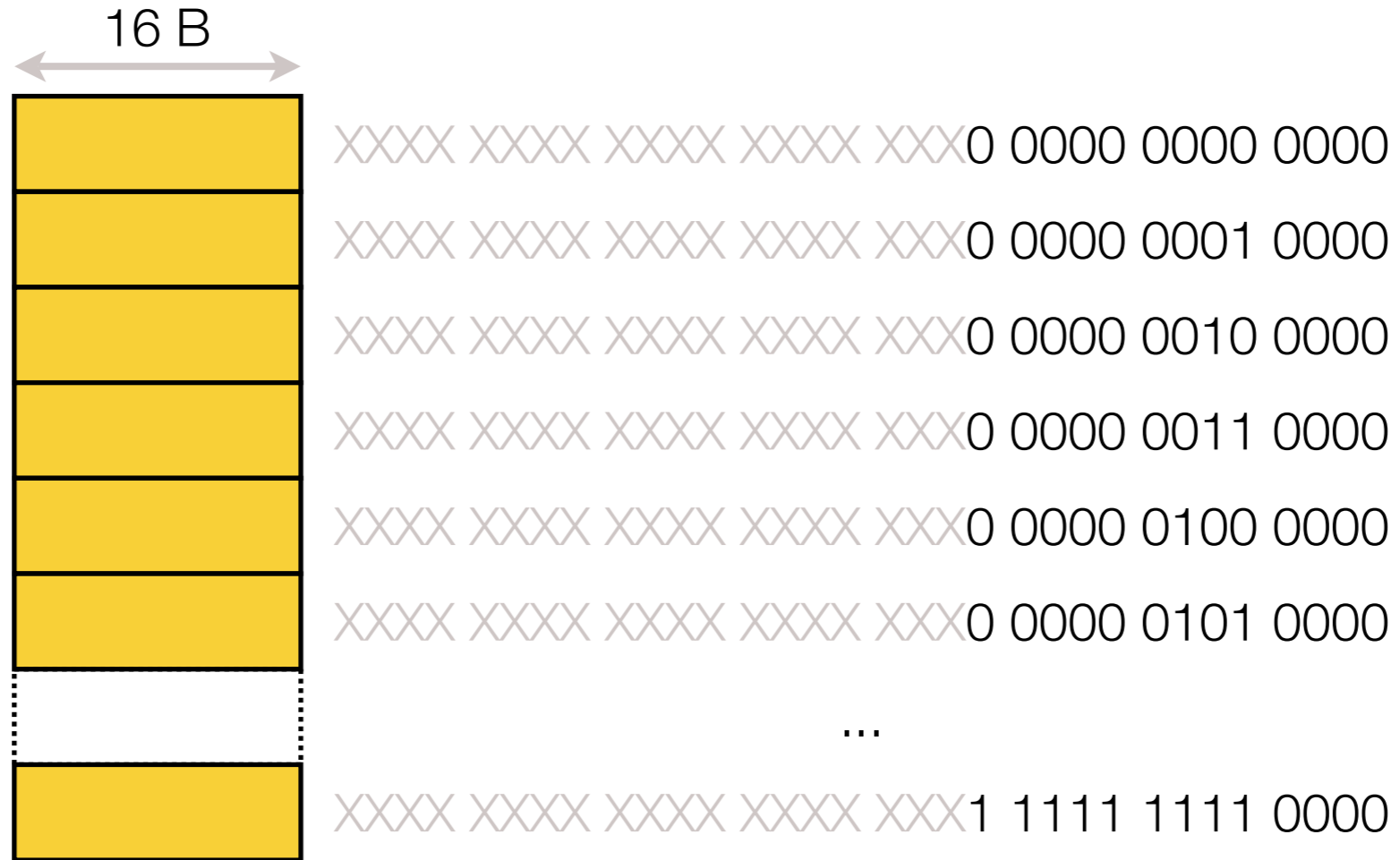
< 25% peak! We evidently still have a lot of work to do...

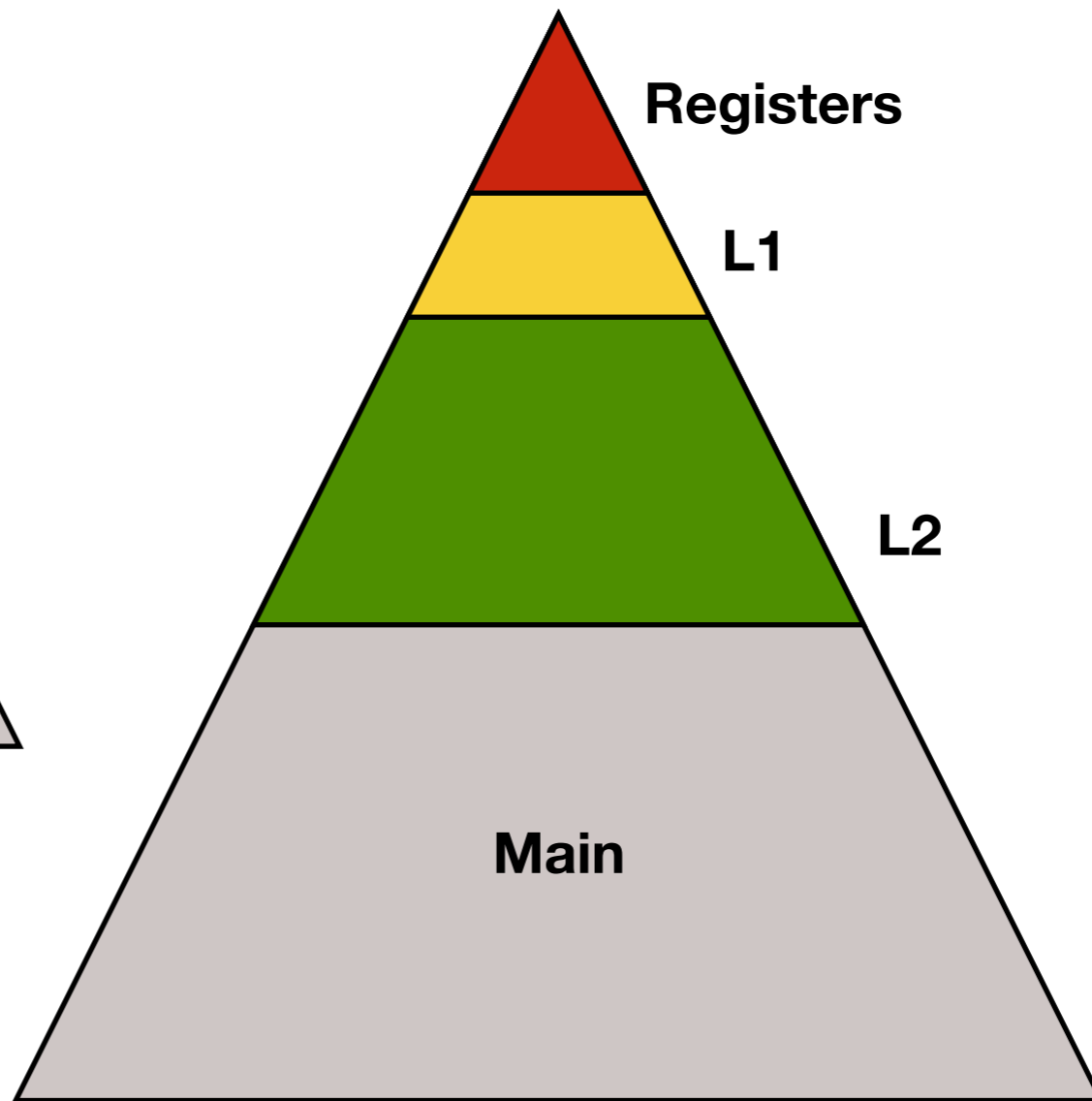
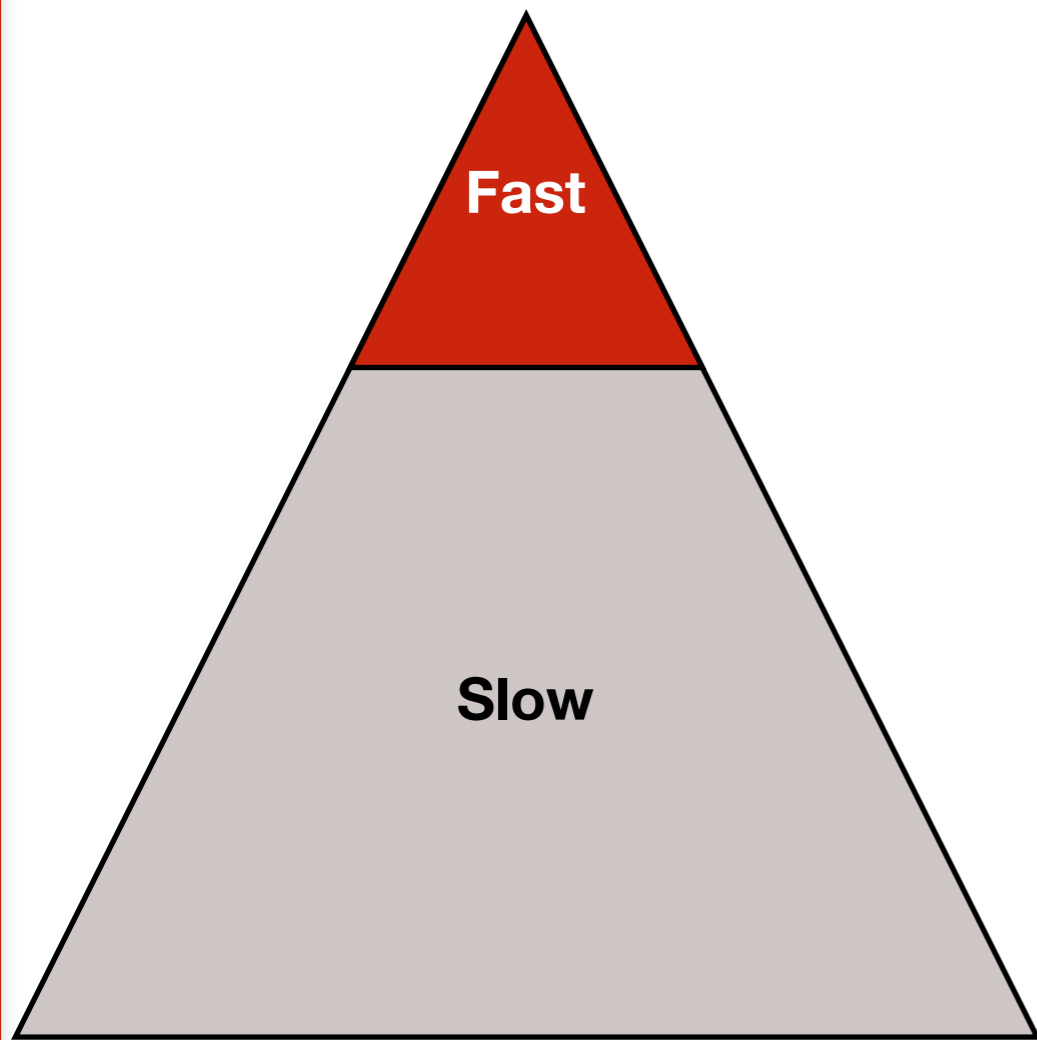


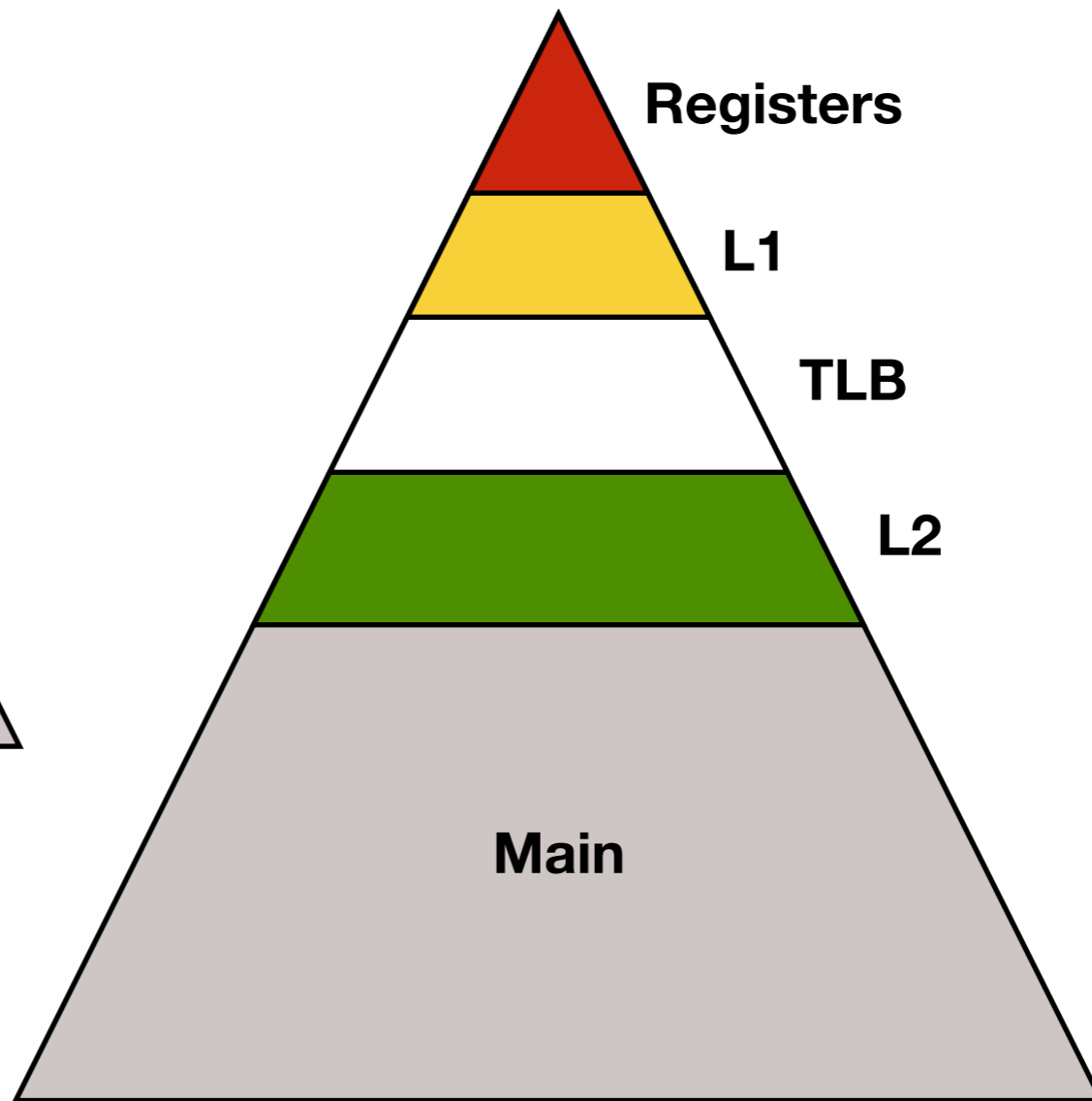
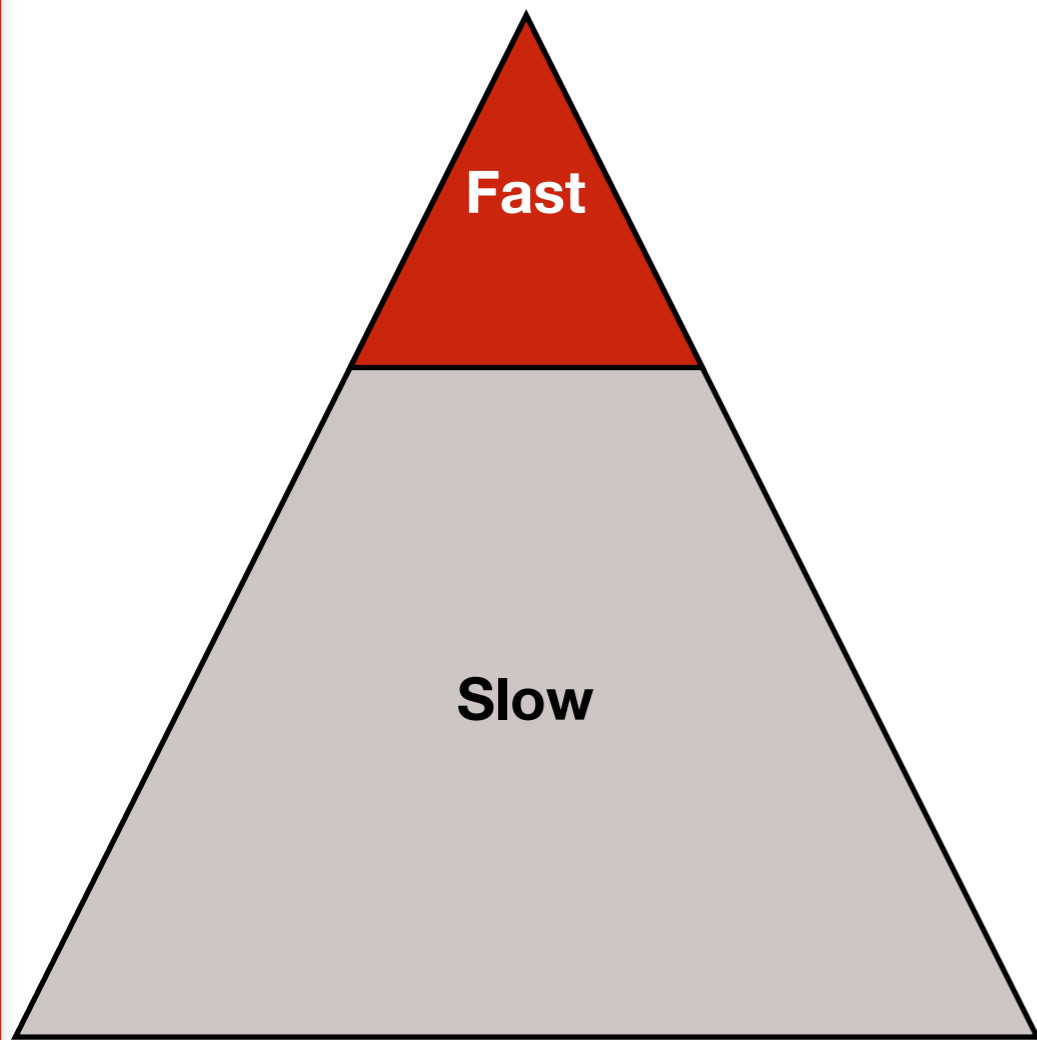
Review: Real memory hierarchies

What happened at powers of 2?

- Byte addressable
- 32-bit addresses
- Cache
 - Direct-mapped
 - 8 KB capacity
 - 16-byte lines







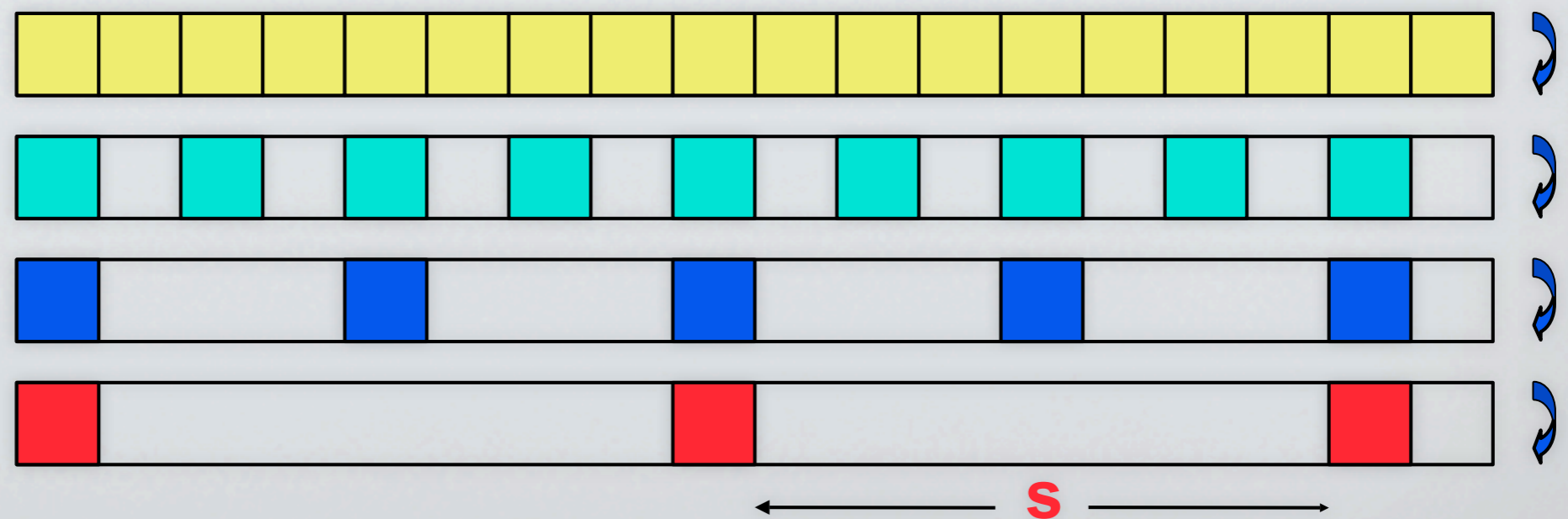


TLB is part of the memory hierarchy

- Translation Look-aside Buffer (TLB) for virtual address space management
 - Divide address space into pages (4—32 KB typical, larger possible)
 - Page table maps virtual to physical addrs & whether page in mem or on disk
 - Page table can be large; TLB caches recent translations
 - May be set-associative or fully-associative
- Conceptually like a cache with large block size, *i.e.*, 1 page
 - May have multiple levels of TLB, just like cache
 - Can prefetch to hide cache misses, but not TLB misses

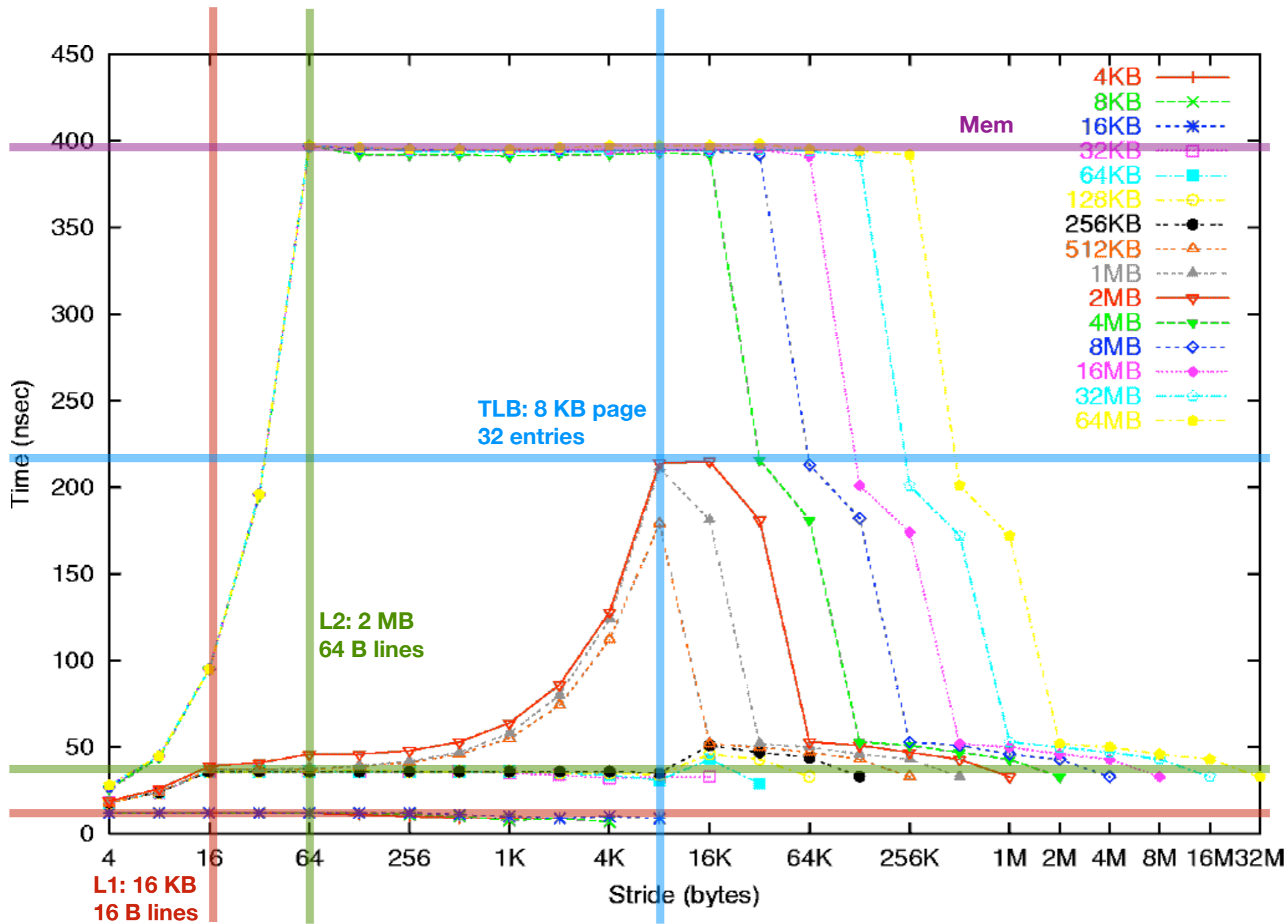


Experiment to observe memory parameters.

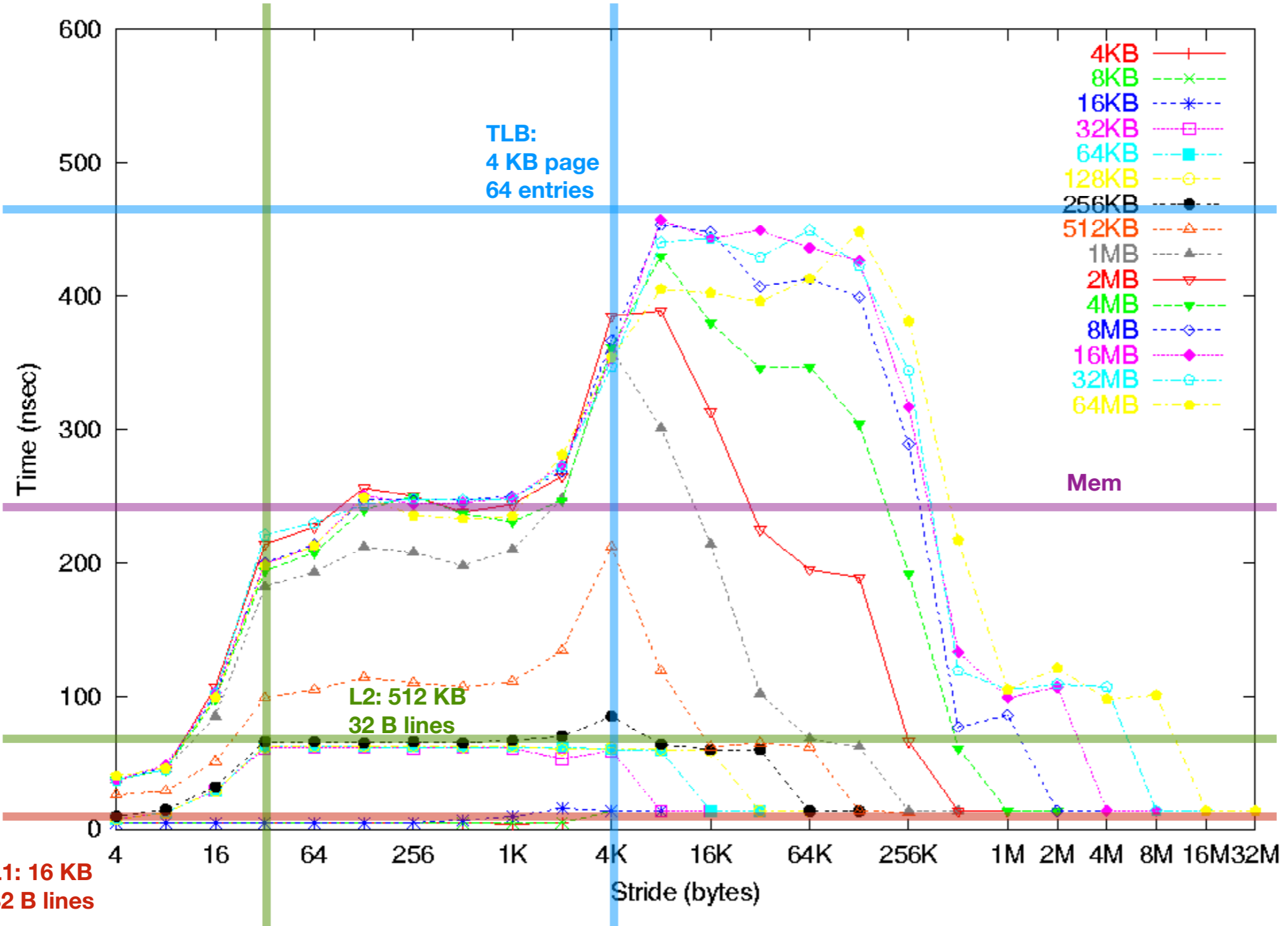


Strided-stream through array; measure average access time.
(Saavedra-Barrera benchmark)

Average Memory Access Time (Saavedra-Barerra) – Sun Ultra Ili (333 MHz)



Average Memory Access Time (Saavedra-Barerra) – Pentium III (Katmai; 550 MHz)



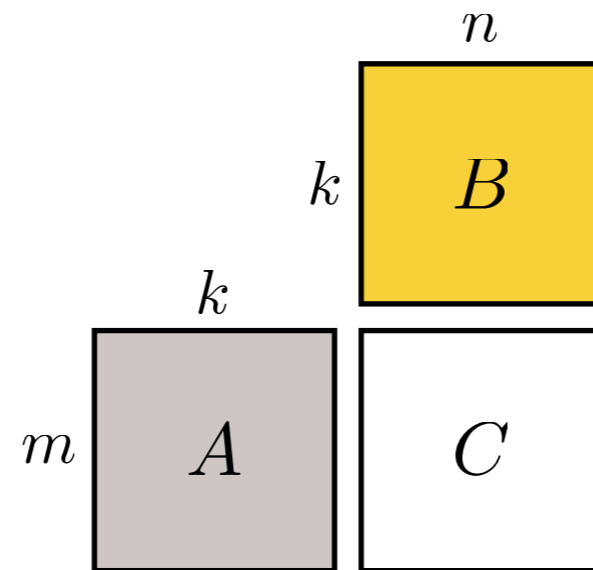


General multi-level blocking

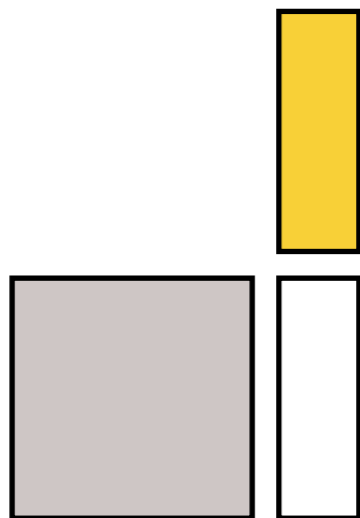
[Goto & van de Geijn (2006)]

$$C \leftarrow C + A \cdot B$$

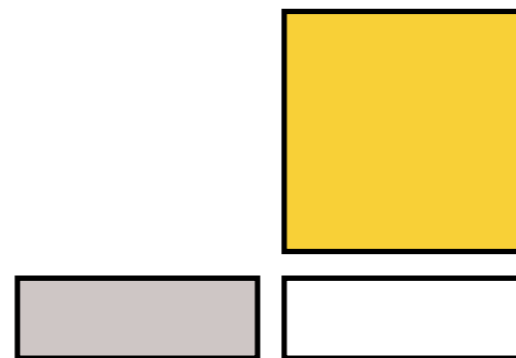
“Matrix-matrix”



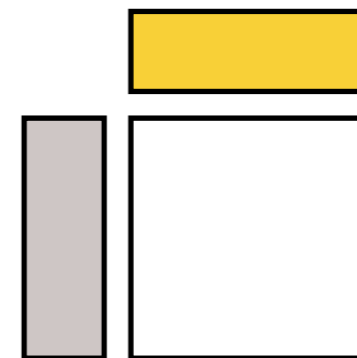
“Matrix-panel”

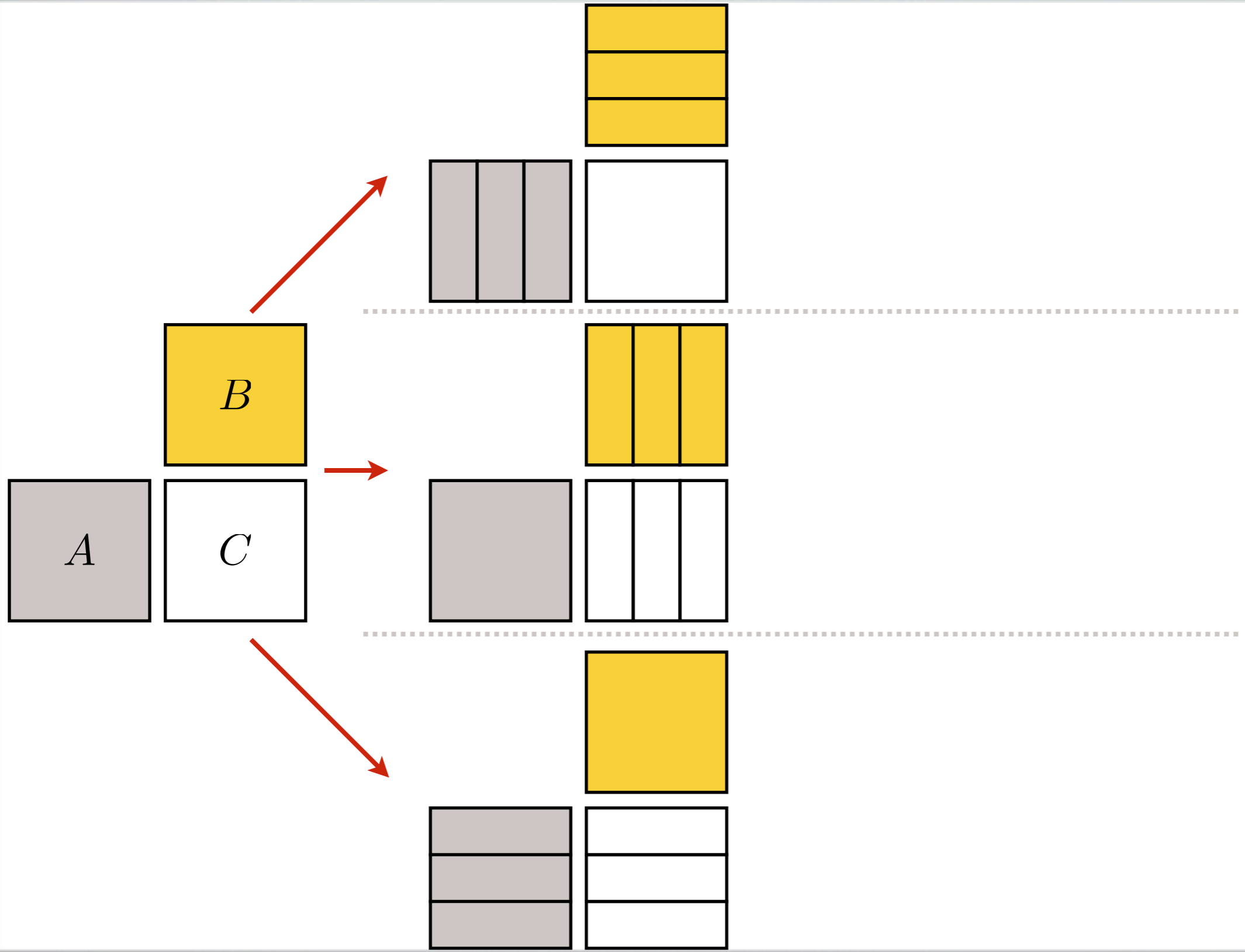


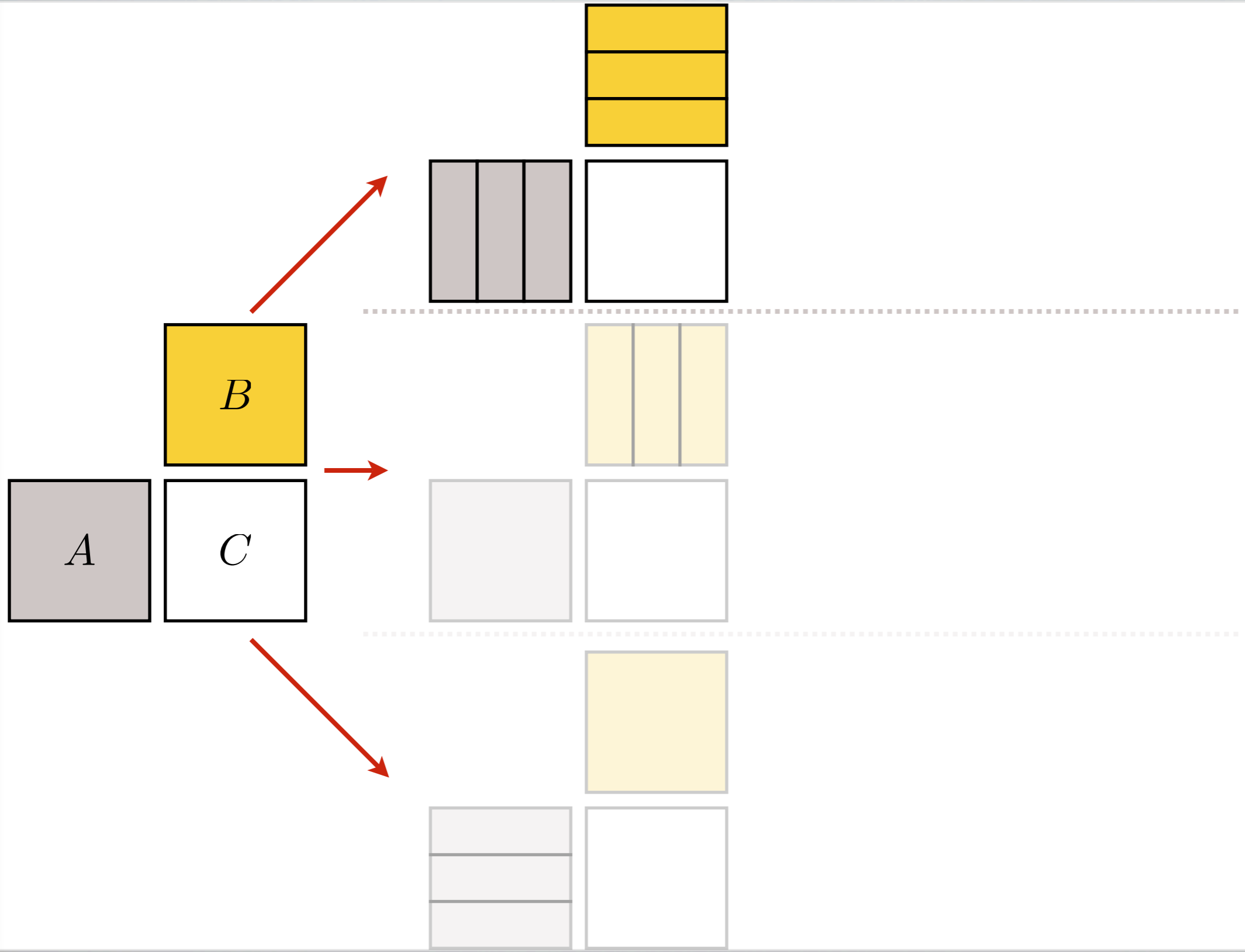
“Panel-matrix”

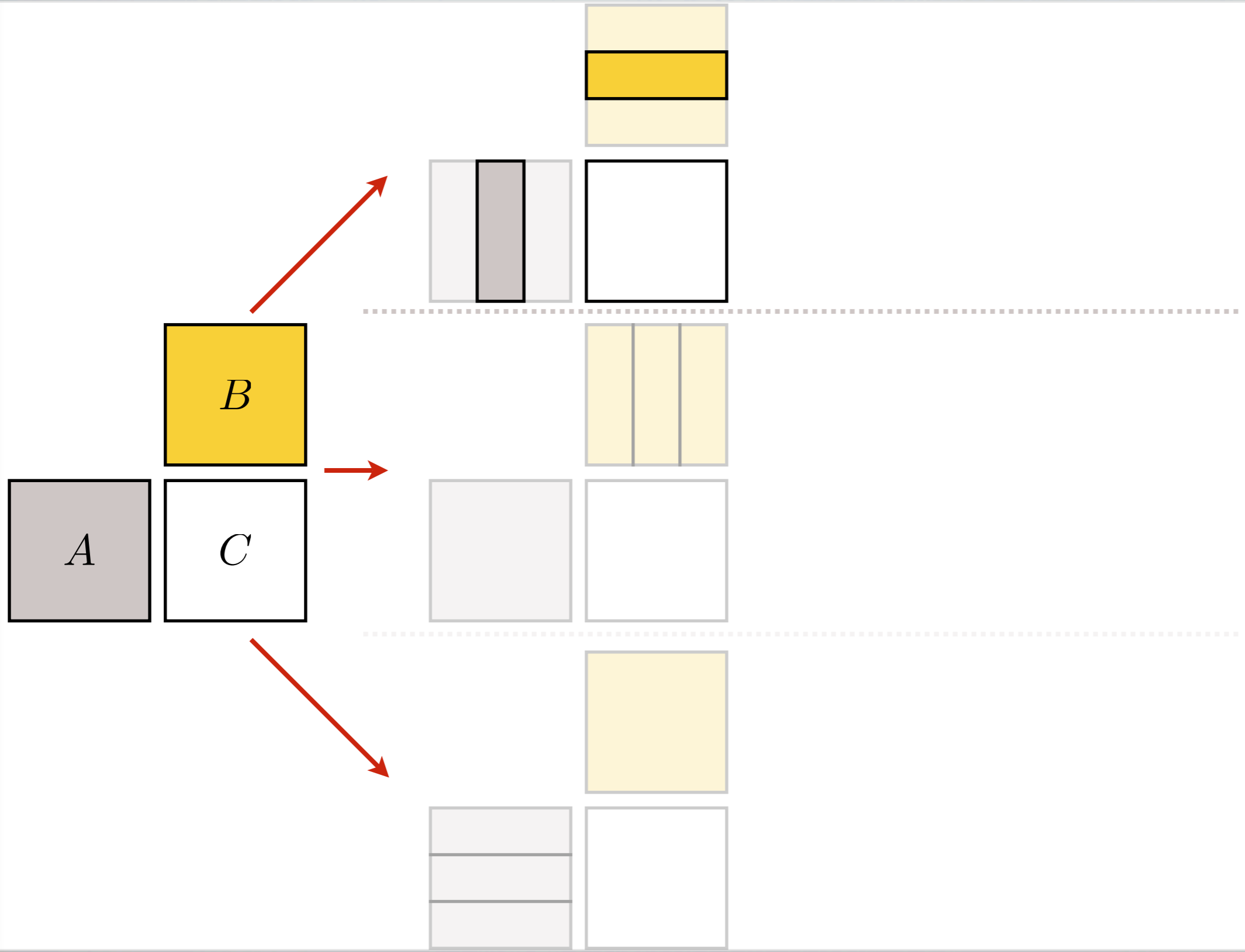


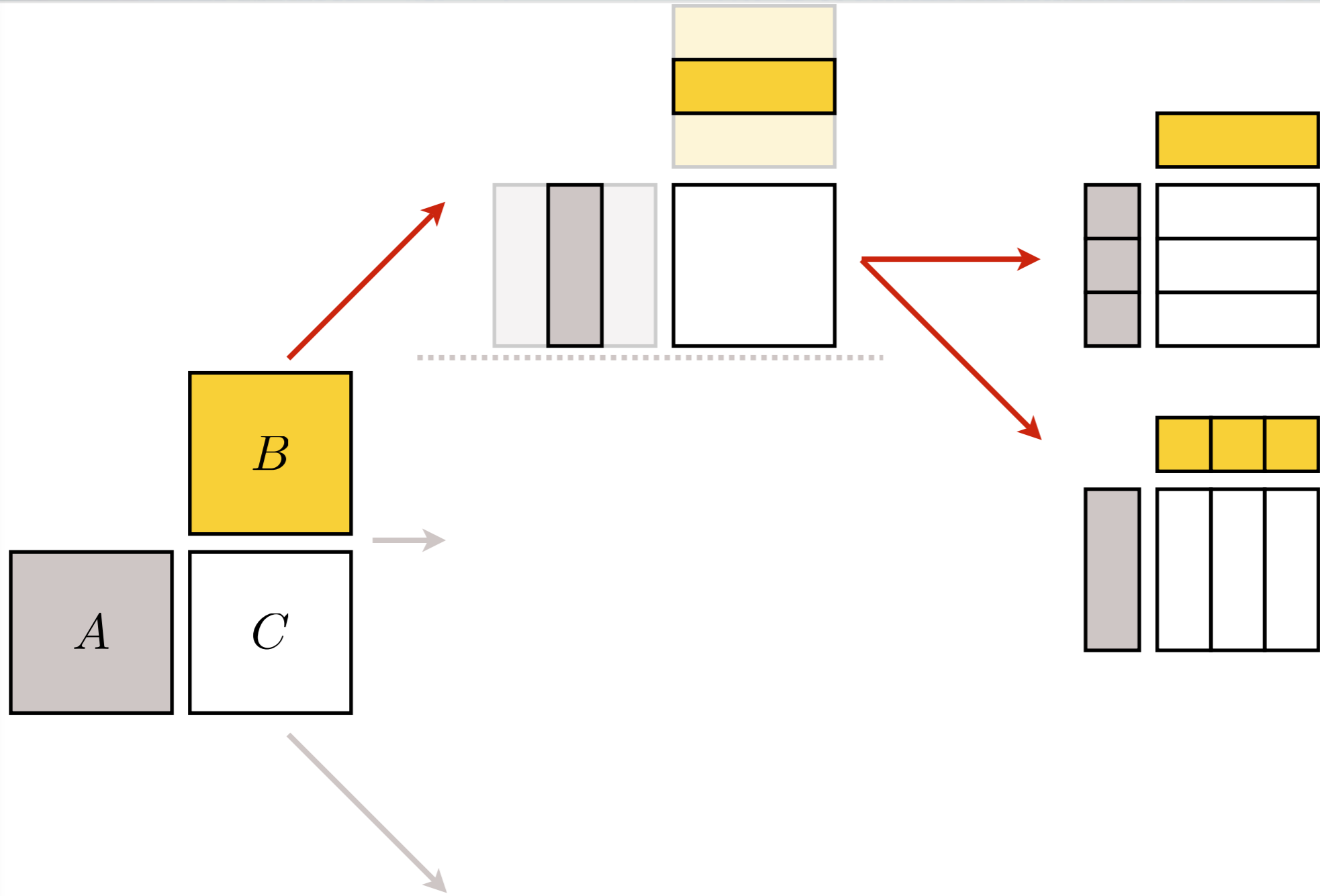
**“Panel-Panel”
or “Fat Outer Product”**





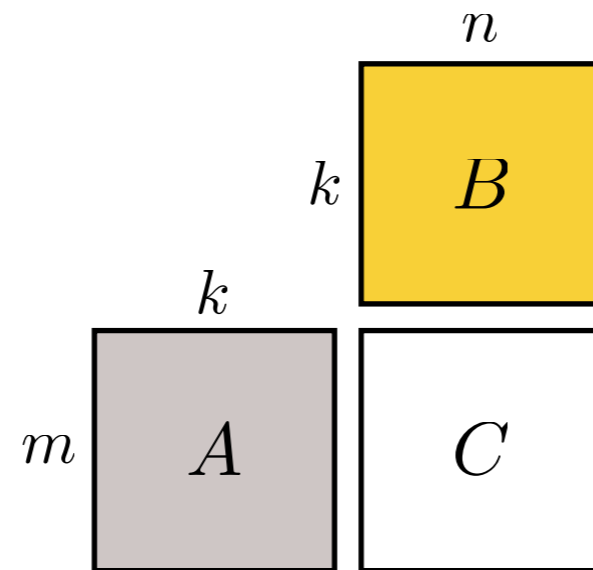




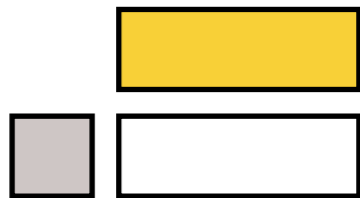


$$C \leftarrow C + A \cdot B$$

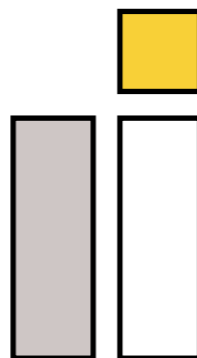
“Matrix-matrix”



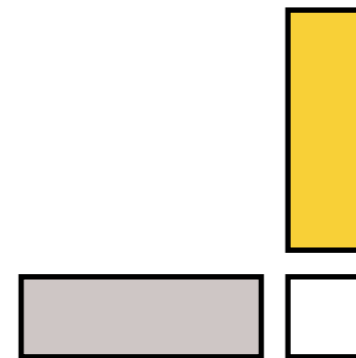
“Block-Panel”

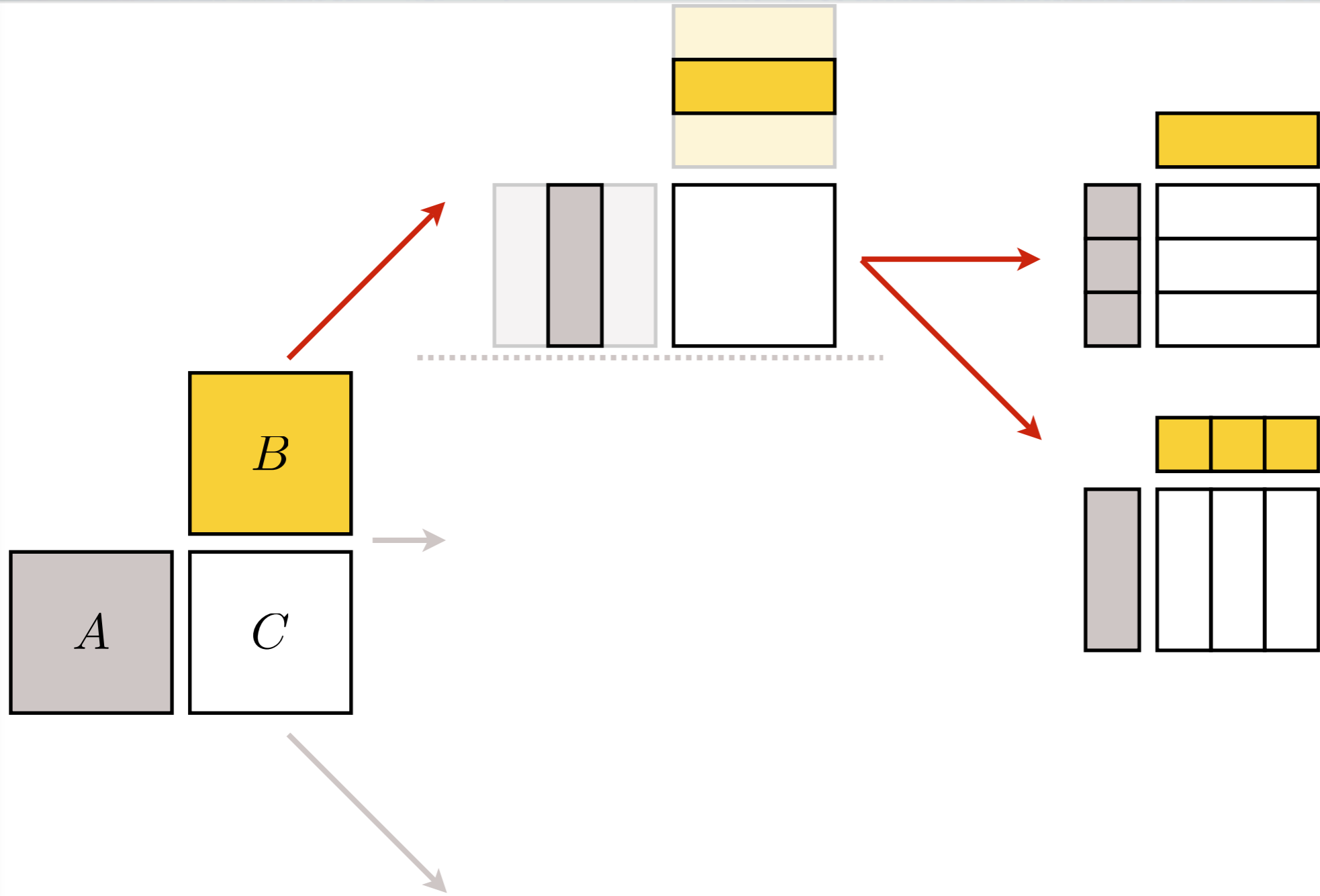


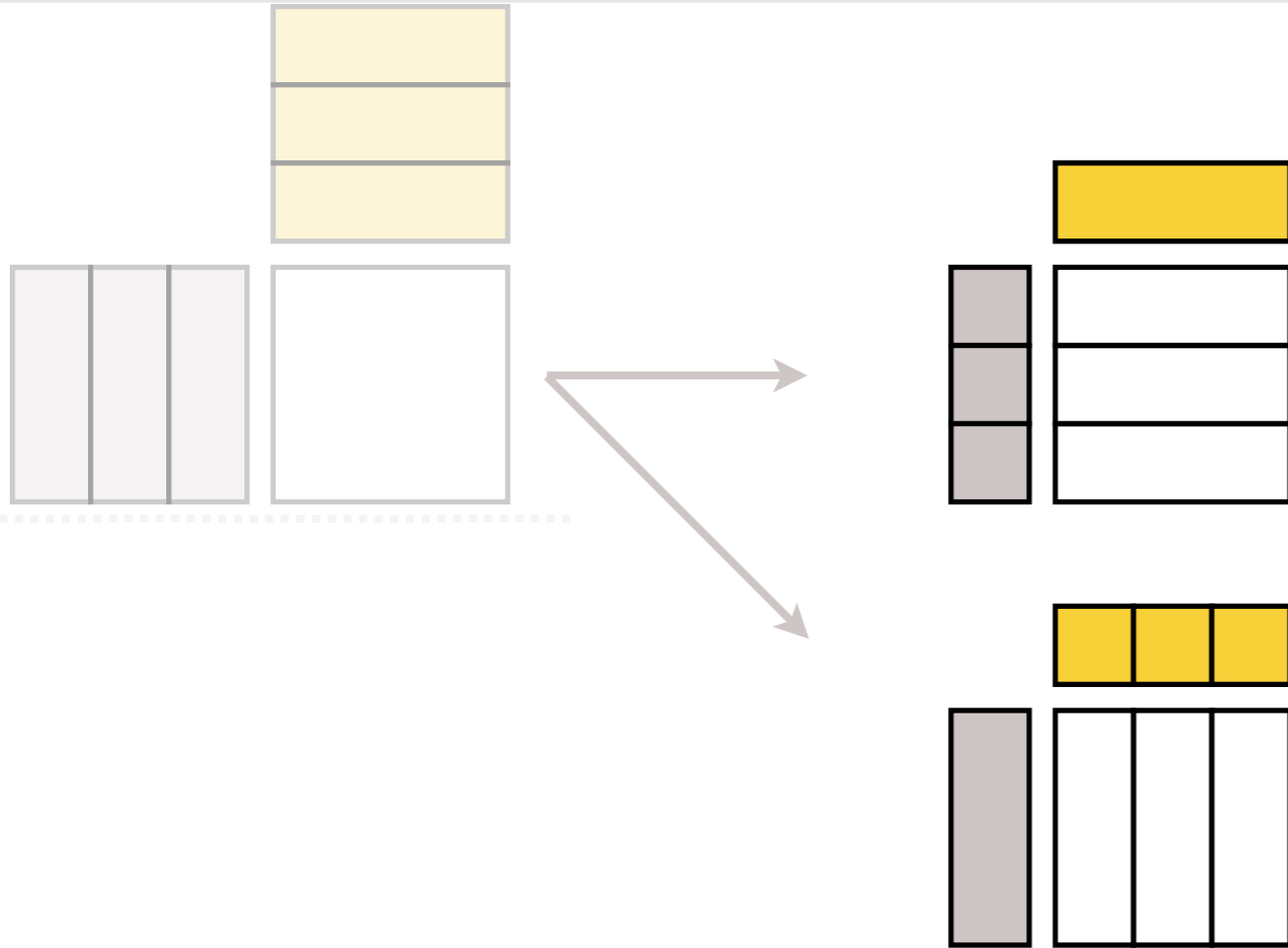
“Panel-block”

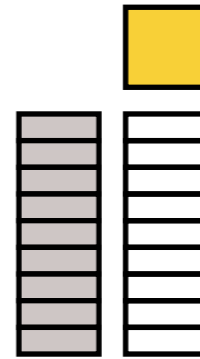
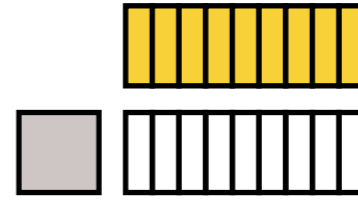
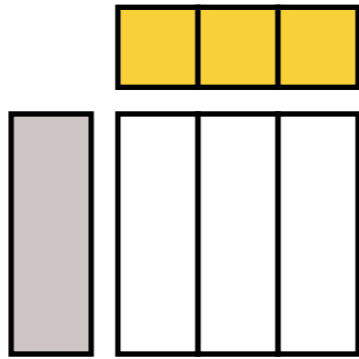
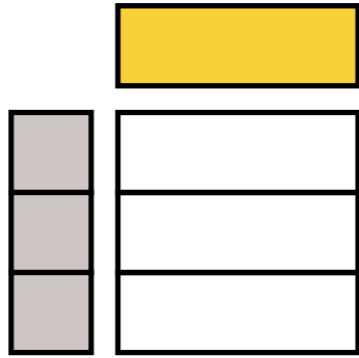
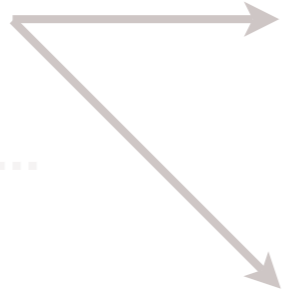
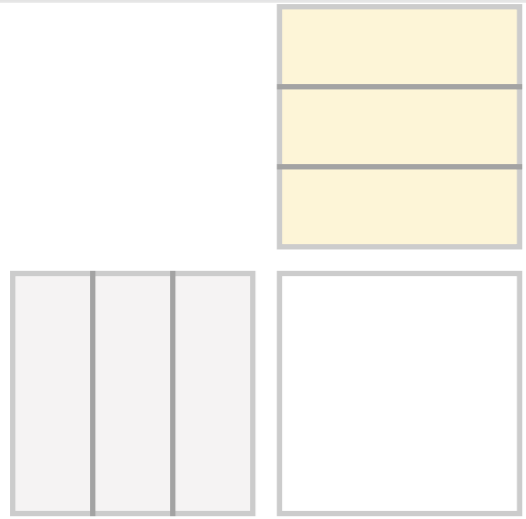


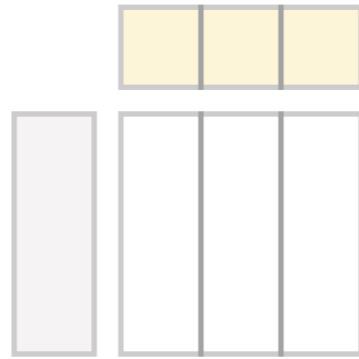
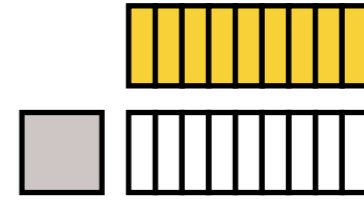
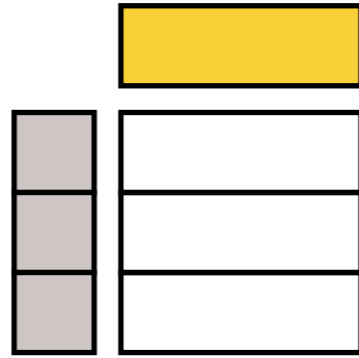
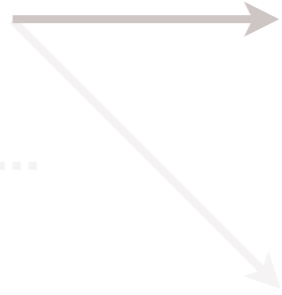
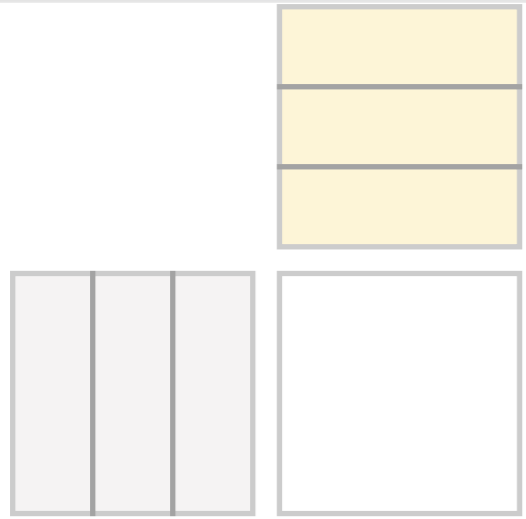
“Fat Dot Product”

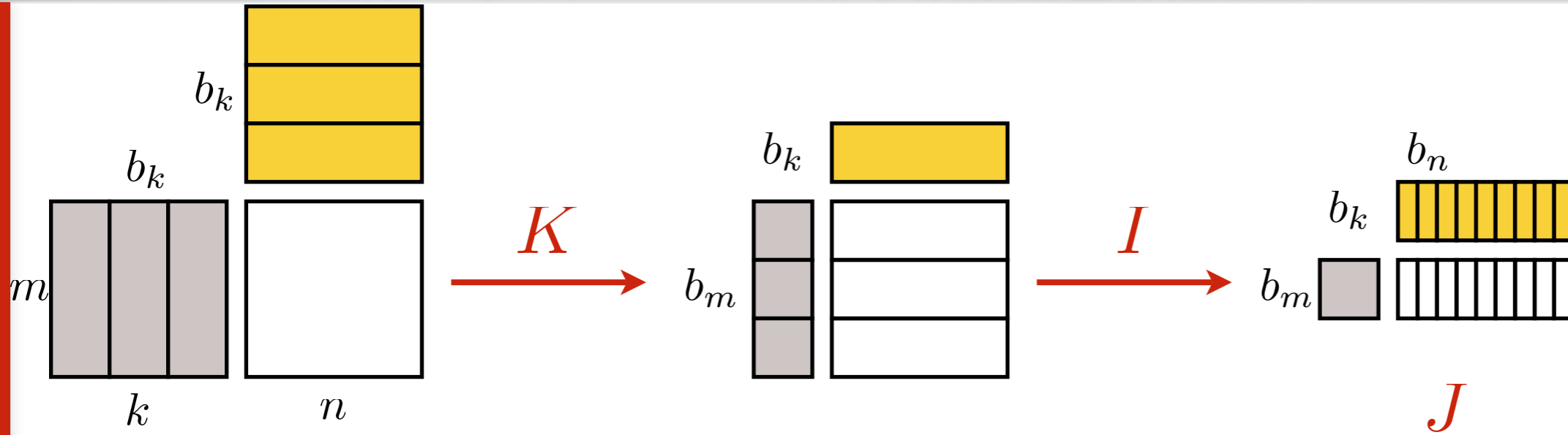












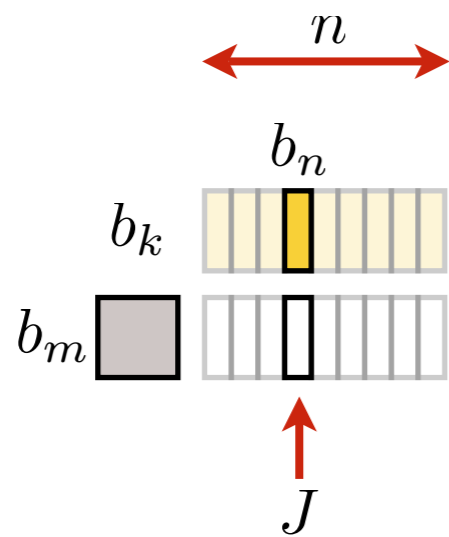
// Let I, J, K = blocks of indices

for $K \leftarrow$ blocks 1 to $\frac{k}{b_k}$ do

for $I \leftarrow$ blocks 1 to $\frac{m}{b_m}$ do

for $J \leftarrow$ blocks 1 to $\frac{n}{b_n}$ do

$$C_{IJ} \leftarrow C_{IJ} + A_{IK} \times B_{KJ}$$



Assumes:

1. A, B_J, C_J fit in cache (e.g., size M)
2. Above \Rightarrow Product runs at peak
3. A not evicted prematurely

$$b_m b_k + (b_k + b_m) b_n \leq M$$

// “Block-panel” multiply

// Load $b_m \times b_k$ block of A into cache

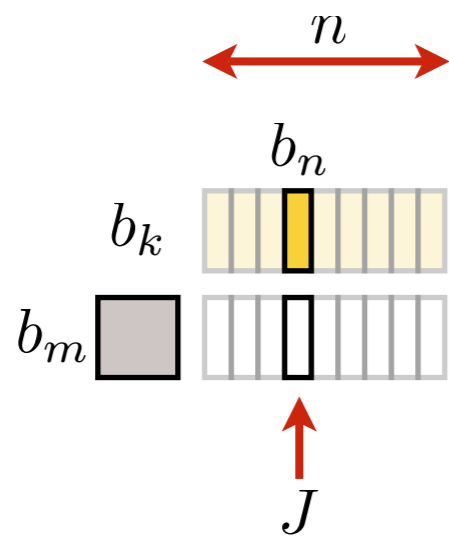
for $J \leftarrow$ blocks 1 **to** $\frac{n}{b_n}$ **do**

// Load $b_k \times b_n$ block of B into cache

// Load $b_m \times b_n$ block of C into cache

$C_J \leftarrow C_J + A \times B_J$

// Store $b_m \times b_n$ block of C to memory



Assumes:

1. A, B_J, C_J fit in cache (e.g., size M)
2. Above \Rightarrow Product runs at peak
3. A not evicted prematurely

$$b_m b_k + (b_k + b_m) b_n \leq M$$

// “Block-panel” multiply

// Load $b_m \times b_k$ block of A into cache

for $J \leftarrow$ blocks 1 to $\frac{n}{b_n}$ do

// Load $b_k \times b_n$ block of B into cache

// Load $b_m \times b_n$ block of C into cache

$C_J \leftarrow C_J + A \times B_J$

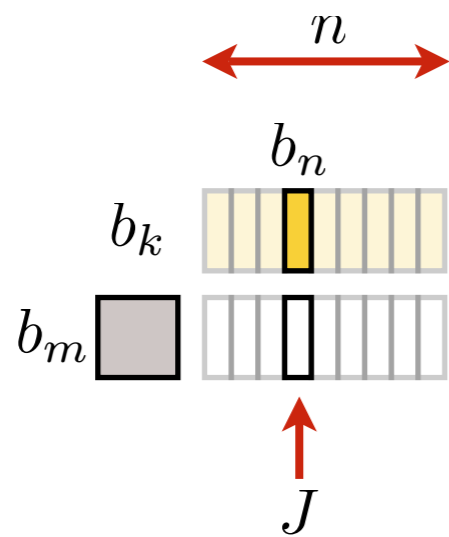
// Store $b_m \times b_n$ block of C to memory

$$f = 2b_m b_k n$$

$$m = b_m b_k + (b_k + 2b_m) n$$

\Downarrow

$$q = \frac{2}{\frac{1}{n} + \left(\frac{1}{b_m} + \frac{2}{b_k} \right)}$$



Given a multi-level memory hierarchy, in what cache should “A” block live?

- ❖ Want large A block
- ❖ L1 cache usually quite small
- ❖ What about L2?

Assumes:

1. A, B_J, C_J fit in cache (e.g., size M)
2. Above \Rightarrow Product runs at peak
3. A not evicted prematurely

$$b_m b_k + (b_k + b_m) b_n \leq M$$

$\rho_1 \equiv$ Peak L1 flop/s

$\beta_2 \equiv$ Peak L2 to CPU bw

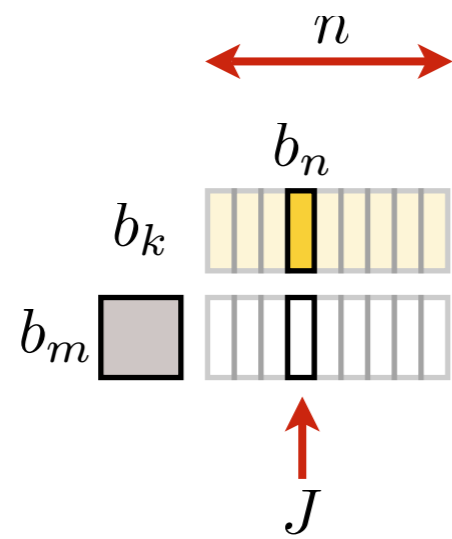
$$\frac{2b_m b_k b_n}{\rho_1} \geq \frac{b_m b_k}{\beta_2}$$

\Downarrow

$$b_n \geq \frac{\rho_1}{2\beta_2}$$

Typically, need $b_n \geq 2$.





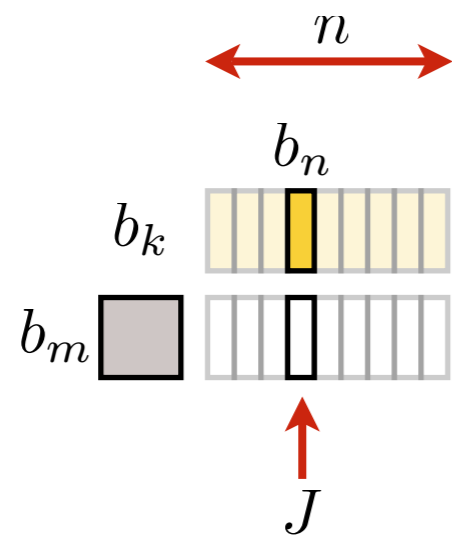
Assumes:

1. A, B_J, C_J fit in cache (e.g., size M)
2. Above \Rightarrow Product runs at peak
3. A not evicted prematurely

$$b_m b_k + (b_k + b_m) b_n \leq M$$

$$b_n \geq \frac{\rho_1}{2\beta_2}$$





Assumes:

1. A, B_J, C_J fit in cache (e.g., size M)
2. Above \Rightarrow Product runs at peak
3. A not evicted prematurely

$$b_m b_k + (b_k + b_m) b_n \leq M$$

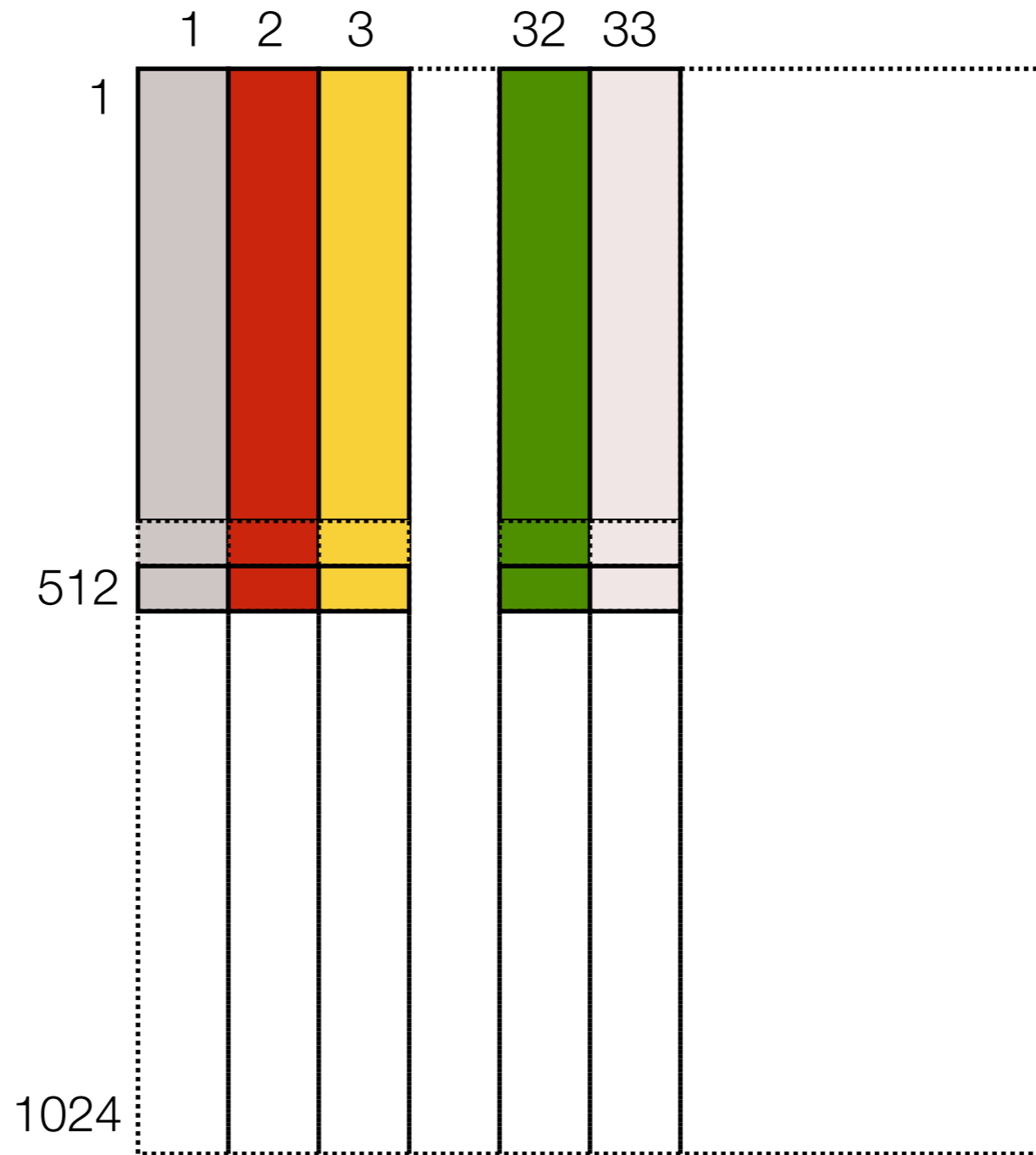
$$b_n \geq \frac{\rho_1}{2\beta_2}$$

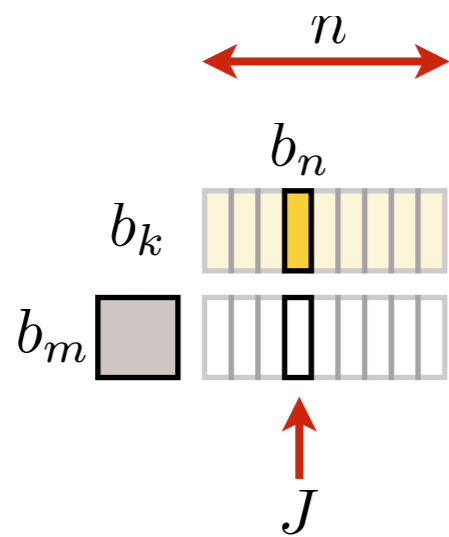
What about the TLB?



Considerations for TLB

- Matrix
 - n = 1024
 - Column-major order
- TLB
 - Page = 4 KB
 - 32 entries





Assumes:

1. A, B_J, C_J fit in cache (e.g., size M)
2. Above \Rightarrow Product runs at peak

3. A not evicted prematurely

4. Operands “fit in” TLB

$$b_m b_k + (b_k + b_m) b_n \leq M$$

$$b_n \geq \frac{\rho_1}{2\beta_2}$$

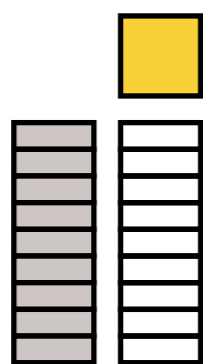
What about the TLB?

Block of A straddles pages, so re-pack on-the-fly \Rightarrow **“Copy optimization”**

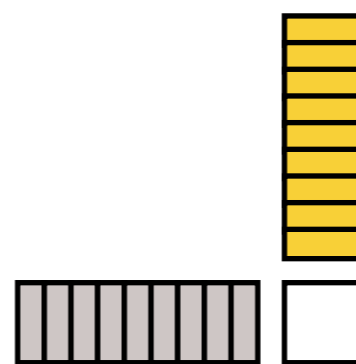
Copy B panel as well

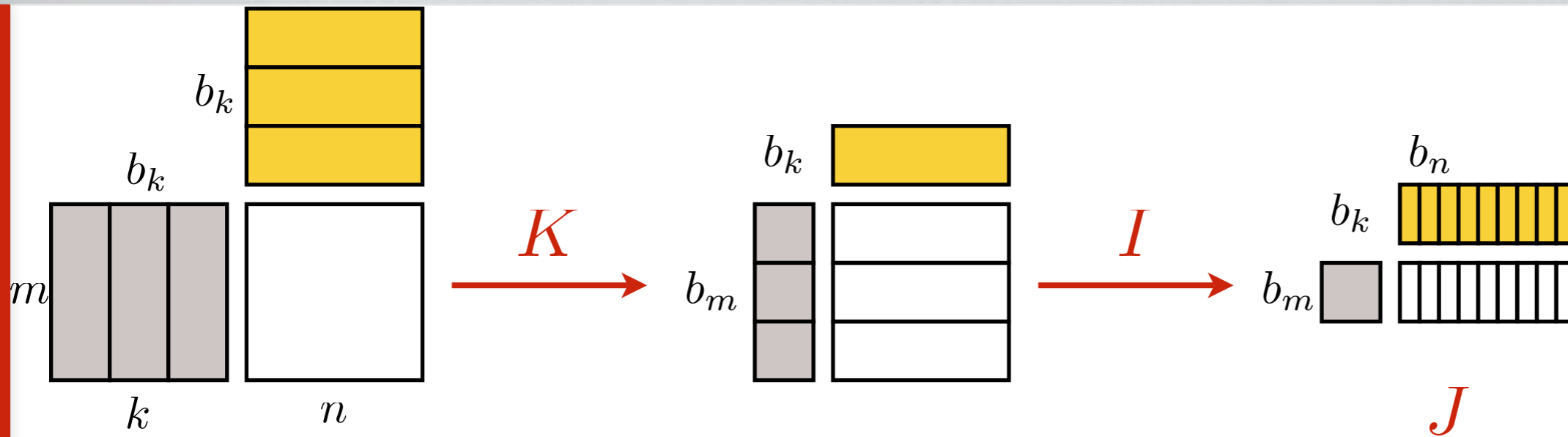


Panel-Block



Fat-Dot





// Let I, J, K = blocks of indices

for $K \leftarrow$ blocks 1 to $\frac{k}{b_k}$ do

$\tilde{B} \leftarrow B_{K,*}$

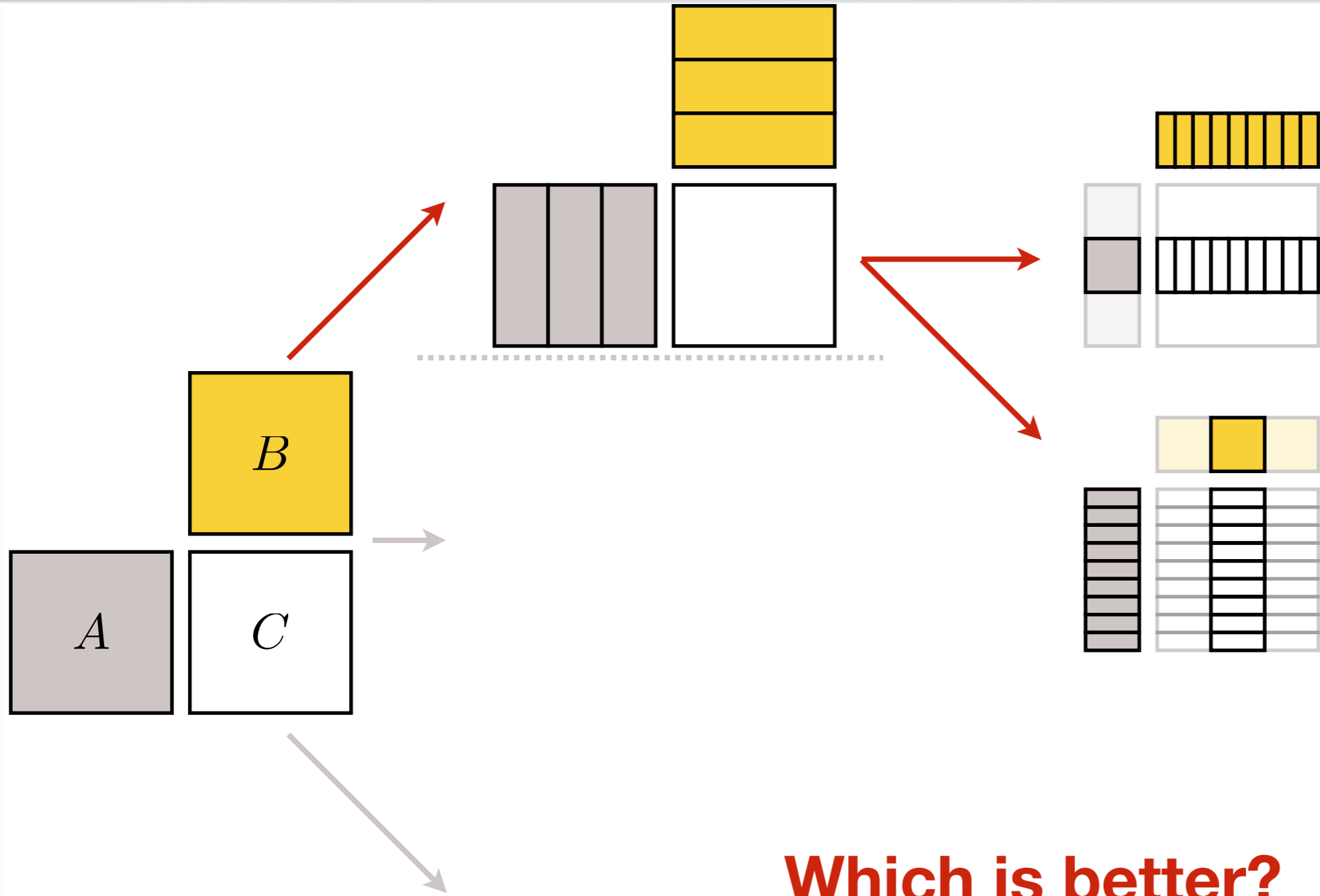
for $I \leftarrow$ blocks 1 to $\frac{m}{b_m}$ do

$\tilde{A} \leftarrow A_{IK}$

for $J \leftarrow$ blocks 1 to $\frac{n}{b_n}$ do

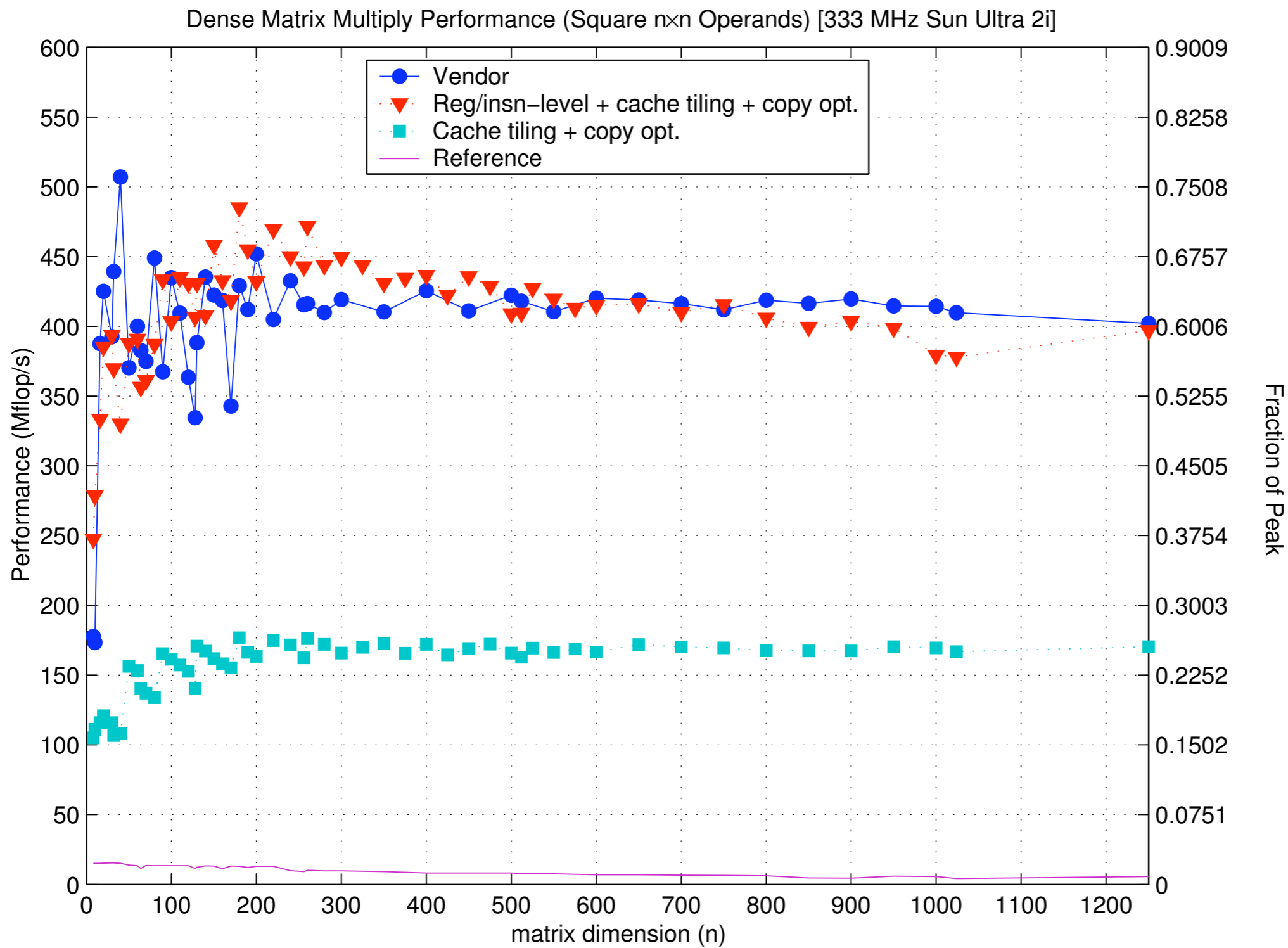
$\tilde{C} \leftarrow \tilde{A} \times \tilde{B}_J$ // Compute in buffer, \tilde{C}

$C_{IJ} \leftarrow C_{IJ} + \tilde{C}$ // Unpack \tilde{C}



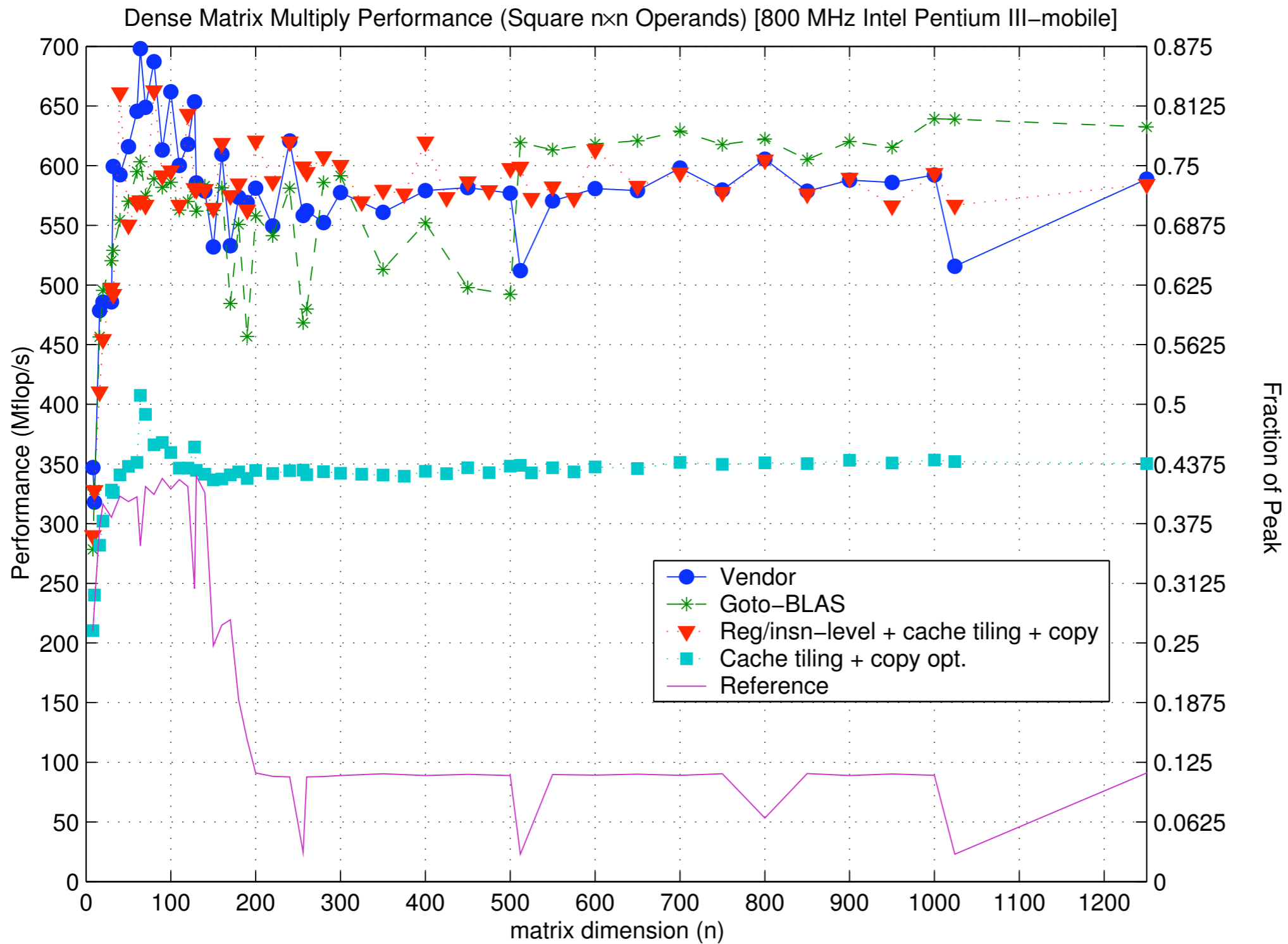
Which is better?



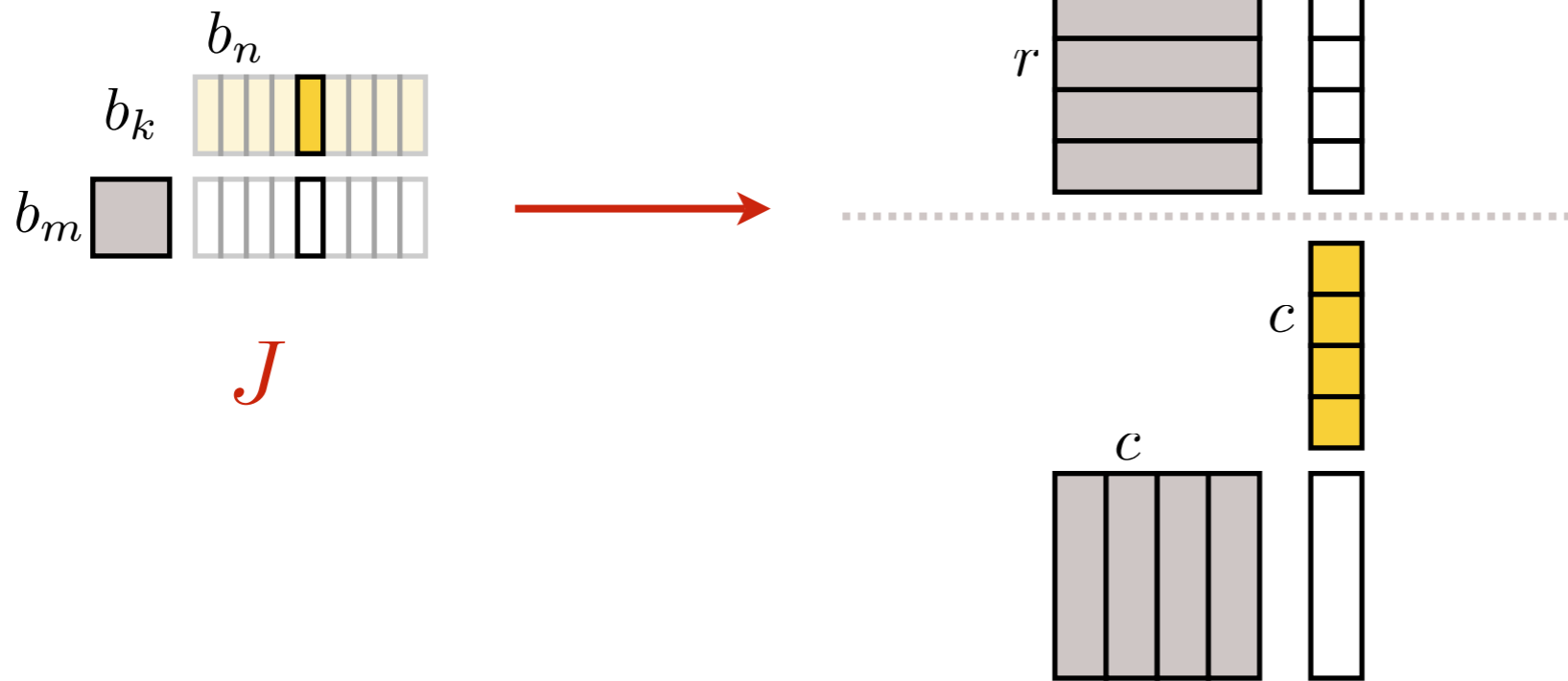


Source: Vuduc, Demmel, Bilmes (IJHPCA 2004)





Source: Vuduc, Demmel, Bilmes (IJHPCA 2004)



Inner-kernel

- Scheduling
- Register allocation



Administrivia



Two joint classes with CS 8803 SC

- **Tues 2/19:** Floating-point issues in parallel computing by me
- **Tues 2/26:** GPGPUs by Prof. Hyesoon Kim
 - **Scribe?**
- **Both classes meet in Klaus 1116E**



Homework 1:

Parallel conjugate gradients

- **Extension:** Due Wednesday 2/27 @ 8:30 am
- Implement a parallel solver for $Ax = b$ (serial C version provided)
 - Evaluate on three matrices: 27-pt stencil, and two application matrices
 - “Simplified:” No preconditioning
- **Performance models to understand scalability of your implementation**
 - Make measurements
 - Build predictive models
- Collaboration encouraged: Compare programming models or platforms




Administrative stuff

- **New room** (dumpier, but cozier?): College of Computing Building **(CCB) 101**
- **Accounts**: Apparently, you already have them
- Front-end login node: **ccil.cc.gatech.edu** (CoC Unix account)
 - We “own” **warp43—warp56**
 - Some docs (**MPI**): <http://www-static.cc.gatech.edu/projects/ihpcl/mpi.html>
 - **Sign-up** for mailing list: <https://mailman.cc.gatech.edu/mailman/listinfo/ihpc-lab>



Projects

- Your goal should be to do something useful, interesting, and/or publishable!
 - Something you're already working on, suitably adapted for this course
 - Faculty-sponsored/mentored
 - Collaborations encouraged



My criteria for “approving” your project

- “Relevant to this course:” Many themes, so think (and “do”) broadly
 - Parallelism and architectures
 - Numerical algorithms
 - Programming models
 - Performance modeling/analysis



General styles of projects

- Theoretical: Prove something hard (high risk)
- Experimental:
 - Parallelize something
 - Take existing parallel program, and improve it using models & experiments
 - Evaluate algorithm, architecture, or programming model

Examples

- *Anything of interest to a faculty member/project outside CoC*
- Parallel sparse triple product ($R^*A^*R^T$, used in multigrid)
- Future FFT
- Out-of-core or I/O-intensive data analysis and algorithms
- Block iterative solvers (convergence & performance trade-offs)
- Sparse LU
- Data structures and algorithms (trees, graphs)
- Look at mixed-precision
- Discrete-event approaches to continuous systems simulation
- Automated performance analysis and modeling, tuning
- “Unconventional,” but related
 - Distributed deadlock detection for MPI
 - UPC language extensions (dynamic block sizes)
 - Exact linear algebra





Inner-kernel



Doesn't the compiler do scheduling and reg. allocation?

- Theorem (Motwani, *et al.*, 1995): Given a DAG, finding the **schedule and register assignment** to minimize register spills is **NP-Hard**.
- Theorem (Belady, 1966): Given a DAG and a schedule, finding the **register assignment** to minimize register spills can be done in \approx **linear time**.

Loop unrolling: Reducing loop overheads

```
for (i=0; i < N; i++)  
    dot += X[i] * Y[i];
```

⇓

```
for (i=0; i < N; i += 4) {  
    dot += X[i] * Y[i];  
    dot += X[i+1] * Y[i+1];  
    dot += X[i+2] * Y[i+2];  
    dot += X[i+3] * Y[i+3];  
}  
for (i -= 4; i < N; i++)  
    dot += X[i] * Y[i];
```

```
double *stX = X + (N/4)*4,  
       *stX2 = X + N;  
do {  
    dot += *X * *Y;  
    dot += X[1] * Y[1];  
    dot += X[2] * Y[2];  
    dot += X[3] * Y[3];  
    X += 4; Y += 4;  
} while (X != stX);  
while (X != stX2)  
    dot += *X++ * *Y++;
```

Source: Clint Whaley's code optimization course (UTSA Spring 2007)

Scalar expansion: Removing serial dependencies

```
sum = 0;
do
{
    sum += *X;
    sum += X[1];
    sum += X[2];
    sum += X[3];
    X += 4;
}
while (X != stX);
```

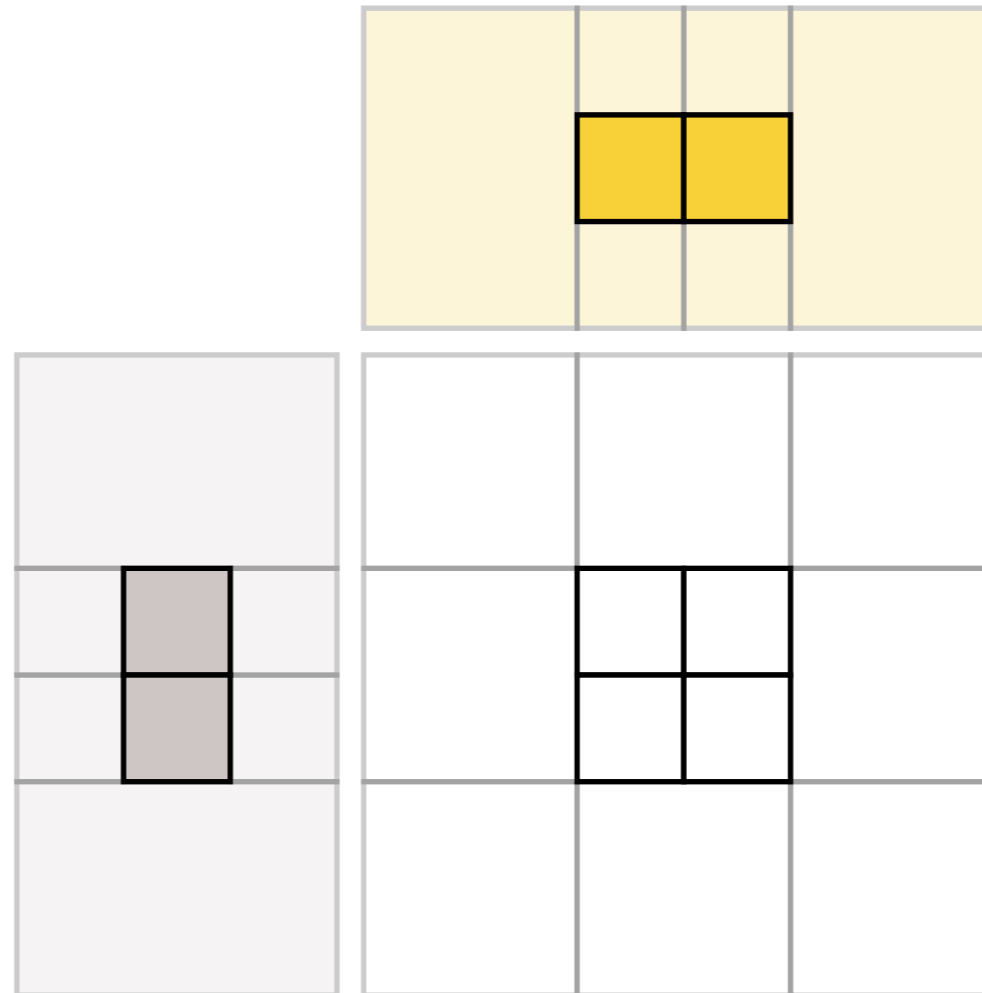
```
sum1 = sum2 = sum3 = sum = 0.0;
do
{
    sum += *X;
    sum1 += X[1];
    sum2 += X[2];
    sum3 += X[3];
    X += 4;
}
while (X != stX);
sum += sum1 + sum2 + sum3;
```

Source: Clint Whaley's code optimization course (UTSA Spring 2007)

Unroll and jam + register blocking

- Reg blk: scalar replacement + register asg

```
for (j=0; j < N; j += 2)
  for (i=0; i < N; i += 2)
  { rC00 = C[i+j*ldc];
    rC10 = C[i+1+j*ldc];
    rC01 = C[i+(j+1)*ldc];
    rC11 = C[i+1+(j+1)*ldc];
    for (k=0; k < N; k++)
    { rB0 = B[k+j*ldb];
      rA0 = A[i+k*lda];
      rB1 = B[k+(j+1)*ldb];
      rA1 = A[i+1+k*lda];
      rC00 += rA0*rB0;
      rC10 += rA1*rB0;
      rC01 += rA0*rB1;
      rC11 += rA1*rB1;
    }
    C[i+j*ldc] = rC00;
    C[i+1+j*ldc] = rC10;
    C[i+(j+1)*ldc] = rC01;
    C[i+1+(j+1)*ldc] = rC11;
  }
```



Source: Clint Whaley's code optimization course (UTSA Spring 2007)

Software pipelining: Interleave iterations to delay dependent instructions

```
for (i=0; i < N; i += 4)  m0 = *X * *Y;    m1 = X[1] * Y[1];
{                          m2 = X[2] * Y[2]; m3 = X[3] * Y[3];
  dot += X[0] * Y[0];      X += 4; Y += 4;
  dot += X[1] * Y[1];      for (i=4; i < N; i += 4)
  dot += X[2] * Y[2];      {
  dot += X[3] * Y[3];      dot += m0;  m0 = X[0] * Y[0];
  X += 4; Y += 4;          dot1 += m1; m1 = X[1] * Y[1];
                          dot2 += m2;  m2 = X[2] * Y[2];
                          dot3 += m3;  m3 = X[3] * Y[3];
                          X += 4; Y += 4;
                          }
}                          dot += m0; dot1 += m1; dot2 += m2; dot3 += m3;
```

The diagram illustrates software pipelining. The original loop body (left) is interleaved with instructions from subsequent iterations (right). Arrows indicate the flow of instructions from iteration $i-4$ and $i-3$ to the pipelined code, and from iteration i and $i+1$ to the pipelined code. The pipelined code shows instructions from iteration i (dot += m0, m0 = X[0] * Y[0]) and iteration $i+1$ (dot1 += m1, m1 = X[1] * Y[1]) interleaved with instructions from iteration $i-4$ (dot += X[0] * Y[0]) and iteration $i-3$ (dot += X[1] * Y[1]).

Source: Clint Whaley's code optimization course (UTSA Spring 2007)

Fetch scheduling, for cache lines and hardware prefetching engines

4 fetches (32 byte CL):

```
for (i=0; i < N; i += 8) {  
    dot0 += x[0] * y[0];  
    dot1 += x[4] * y[4];  
    dot0 += x[1] * y[1];  
    dot1 += x[5] * y[5];  
    dot0 += x[2] * y[2];  
    dot1 += x[6] * y[6];  
    dot0 += x[3] * y[3];  
    dot1 += x[7] * y[7];  
    x += 8; y += 8;  
}
```

4 prefetch units

```
n2 = N>>1;  
xx = X + n2; yy = Y + n2;  
for (i=0; i < n2; i++) {  
    dot0 += x[i] * y[i];  
    dot1 += xx[i] * yy[i];  
}
```

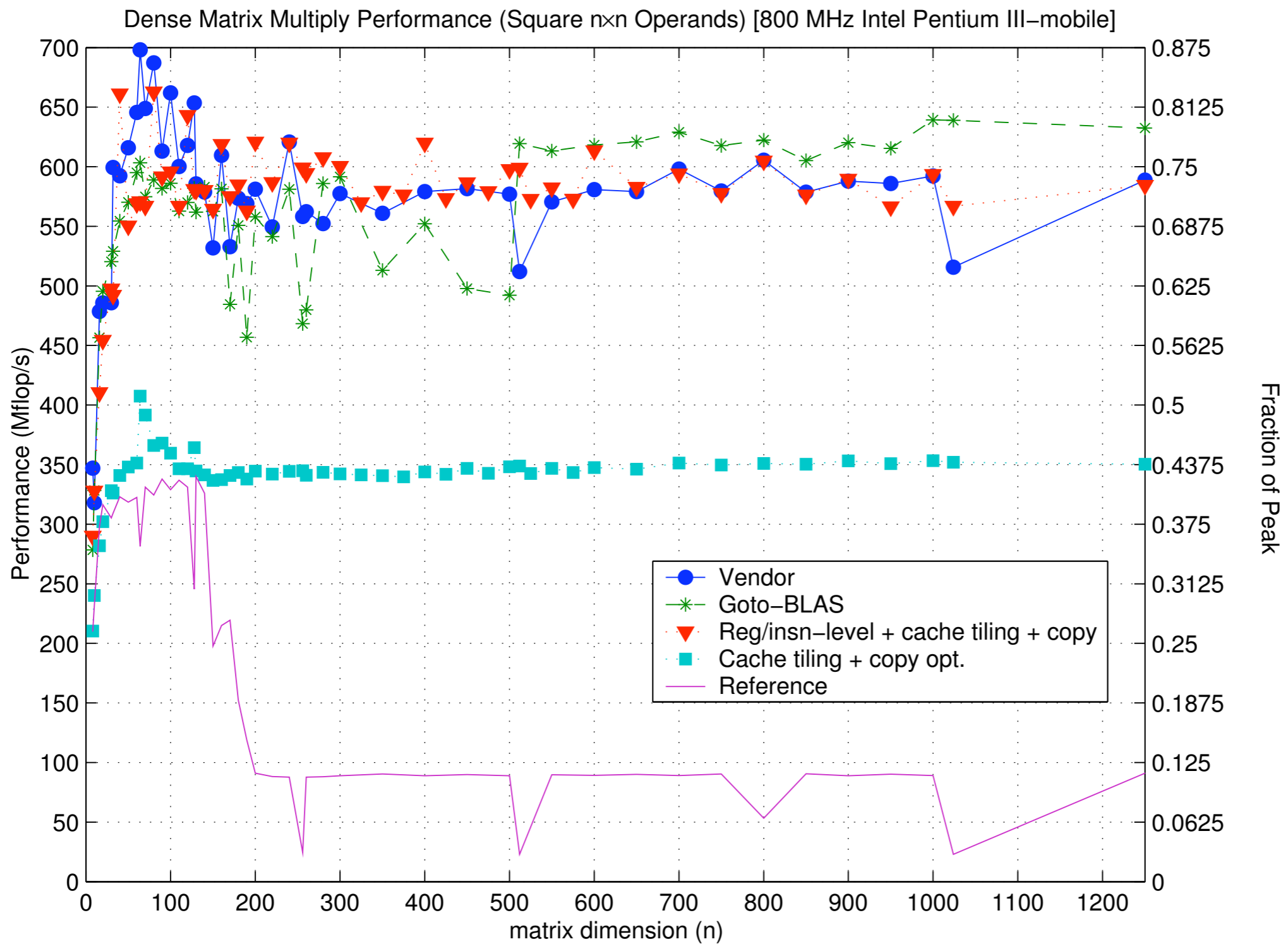
Source: Clint Whaley's code optimization course (UTSA Spring 2007)

Software prefetching

```
#ifndef OPTERON
    #define XDIST 256
#elif defined(P4E)
    #define XDIST 512
#else
    #define XDIST 768
#endif
#define YDIST XDIST

for (i=0; i < N; i += 4) {
    MY_PREF(X+XDIST);
    MY_PREF(Y+YDIST);
    dot0 += X[0] * Y[0];
    dot1 += X[1] * Y[1];
    dot2 += X[2] * Y[2];
    dot3 += X[3] * Y[3];
}
```

Source: Clint Whaley's code optimization course (UTSA Spring 2007)





“In conclusion...”



Backup slides

Saavedra-Barrera Benchmark: Time to execute 1 load [Power3]

