Georgia Tech College of Computing
Computational Science and Engineering

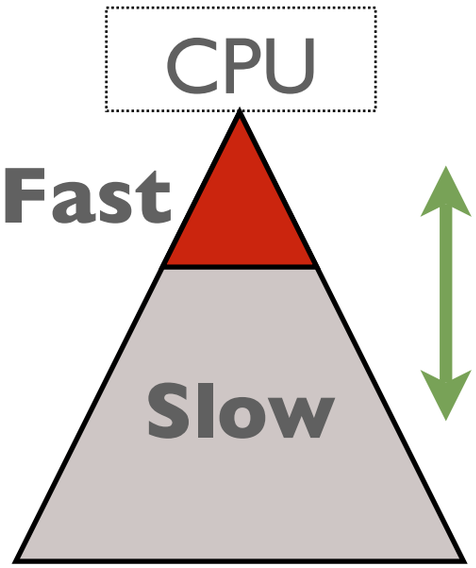
Memory hierarchies (2/2)

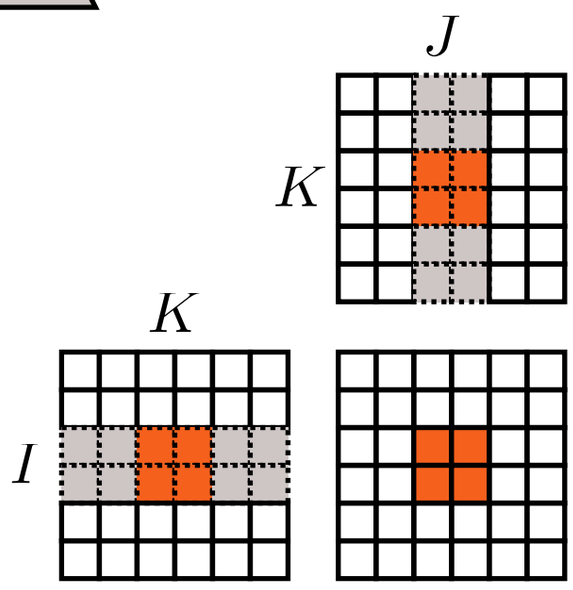
Prof. Rich Vuduc <richie@cc>

Feb. 20, 2009

- Quick recap:
  An algorithmic view of the memory hierarchy

Idea 1: Machine balance & computational intensity.





$$\text{flops} = f \sim n^3$$

$$\text{mops} = m \sim \frac{n^3}{b}$$

$$q = \frac{f}{m} \sim b$$

$$\Downarrow$$

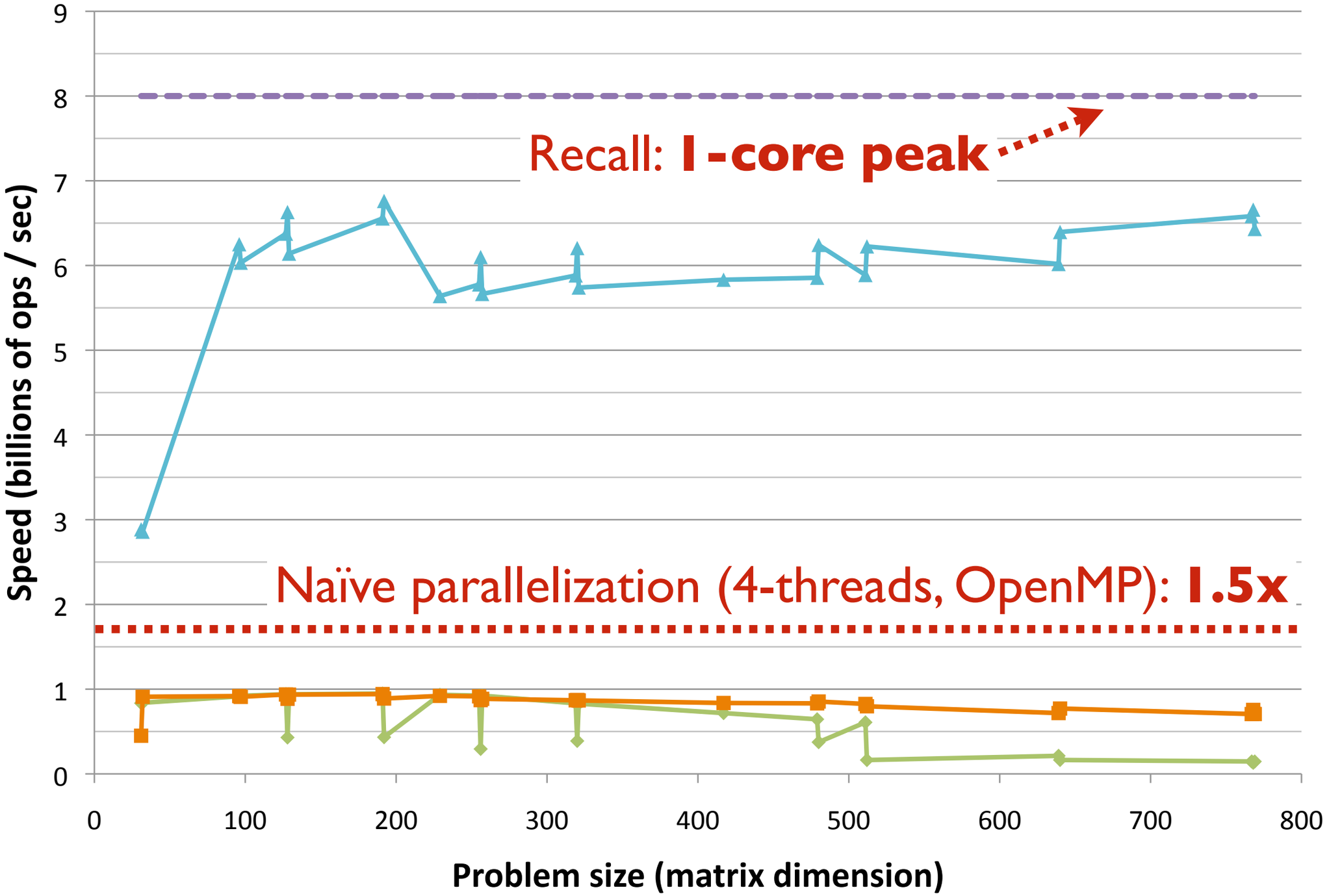
$$\frac{T}{f \cdot \tau} \sim 1 + \frac{\alpha}{\tau} \cdot \frac{1}{b}$$

Idea 2: I/O optimality.

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

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

Theory vs. Practice



Large, Complex Tuning Spaces

K. Jiang + A. Salahuddin (CSE 6230, Fall 2008)

```
#include <stdio.h>
#include <malloc.h>
#include <pmmintrin.h>
#include <string.h>

const char* dgemm_desc = "Tuned dgemm.";

int m;
int n;
int o;
int offc;

int offa;
int offb;

double l1_cache[360000] __attribute__((aligned(16)));
double b_transpose[640000] __attribute__((aligned(16)));
double a_copy[640000] __attribute__((aligned(16)));

double m0_0[160000] __attribute__((aligned(16)));
double m1_0[160000] __attribute__((aligned(16)));
double m2_0[160000] __attribute__((aligned(16)));
double m3_0[160000] __attribute__((aligned(16)));
double m4_0[160000] __attribute__((aligned(16)));
double m5_0[160000] __attribute__((aligned(16)));
double m6_0[160000] __attribute__((aligned(16)));

double a0_0[160000] __attribute__((aligned(16)));
double a1_0[160000] __attribute__((aligned(16)));
double a2_0[160000] __attribute__((aligned(16)));
double a3_0[160000] __attribute__((aligned(16)));
double a4_0[160000] __attribute__((aligned(16)));
double a5_0[160000] __attribute__((aligned(16)));
double a6_0[160000] __attribute__((aligned(16)));

double b0_0[160000] __attribute__((aligned(16)));
double b1_0[160000] __attribute__((aligned(16)));
double b2_0[160000] __attribute__((aligned(16)));
double b3_0[160000] __attribute__((aligned(16)));
double b4_0[160000] __attribute__((aligned(16)));
double b5_0[160000] __attribute__((aligned(16)));
double b6_0[160000] __attribute__((aligned(16)));

double m0_1[65536] __attribute__((aligned(16)));
double m1_1[65536] __attribute__((aligned(16)));
double m2_1[65536] __attribute__((aligned(16)));
double m3_1[65536] __attribute__((aligned(16)));
double m4_1[65536] __attribute__((aligned(16)));
double m5_1[65536] __attribute__((aligned(16)));
double m6_1[65536] __attribute__((aligned(16)));

double a0_1[65536] __attribute__((aligned(16)));
double a1_1[65536] __attribute__((aligned(16)));
double a2_1[65536] __attribute__((aligned(16)));
double a3_1[65536] __attribute__((aligned(16)));
double a4_1[65536] __attribute__((aligned(16)));
double a5_1[65536] __attribute__((aligned(16)));
double a6_1[65536] __attribute__((aligned(16)));

double b0_1[65536] __attribute__((aligned(16)));
double b1_1[65536] __attribute__((aligned(16)));
double b2_1[65536] __attribute__((aligned(16)));
double b3_1[65536] __attribute__((aligned(16)));
double b4_1[65536] __attribute__((aligned(16)));
double b5_1[65536] __attribute__((aligned(16)));
double b6_1[65536] __attribute__((aligned(16)));

double m0_2[16384] __attribute__((aligned(16)));
double m1_2[16384] __attribute__((aligned(16)));
double m2_2[16384] __attribute__((aligned(16)));
double m3_2[16384] __attribute__((aligned(16)));
double m4_2[16384] __attribute__((aligned(16)));
double m5_2[16384] __attribute__((aligned(16)));
double m6_2[16384] __attribute__((aligned(16)));

double a0_2[16384] __attribute__((aligned(16)));
double a1_2[16384] __attribute__((aligned(16)));
double a2_2[16384] __attribute__((aligned(16)));
double a3_2[16384] __attribute__((aligned(16)));
double a4_2[16384] __attribute__((aligned(16)));
double a5_2[16384] __attribute__((aligned(16)));
double a6_2[16384] __attribute__((aligned(16)));

double b0_2[16384] __attribute__((aligned(16)));
double b1_2[16384] __attribute__((aligned(16)));
double b2_2[16384] __attribute__((aligned(16)));
double b3_2[16384] __attribute__((aligned(16)));
double b4_2[16384] __attribute__((aligned(16)));
double b5_2[16384] __attribute__((aligned(16)));
double b6_2[16384] __attribute__((aligned(16)));

double * l1_ptr = l1_cache;

void print_mat(double * output, int size)
{
  int i, j;
  for(i=0; i<size; i++)
  {
    for(j=0; j<size; j++)
    {
      printf("%4lf ", *(output + i*size + j));
    }
    printf("\n");
  }
}

int block_size[133] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
  10, 11, 12, 13, 14, 15, 8, 17, 18, 19,  //10 x = 1
  20, 21, 22, 22, 22, 23, 24, 14, 14, 15,  //20 x = 1,2
  16, 16, 32, 17, 18, 18, 18, 18, 19, 20, 20, //30 x = 2
  20, 14, 14, 16, 16, 16, 16, 16, 16, 18,  //40 x = 2,3
  18, 18, 18, 18, 18, 14, 14, 15, 15, 15,  //50 x = 3,4
  15, 14, 14, 14, 20, 14, 14, 12, 12, 12,  //60 x = 4,5,6
  12, 12, 12, 11, 11, 11, 11, 11, 10, 10,  //70 x = 6,7,8
  16, 12, 14, 11, 11, 11, 11, 11, 10, 10,  //80 x = 8,9
  10, 10, 10, 10, 10, 10, 16, 10, 18, 9,  //90 x = 9,10,11
  9, 9, 9, 9, 9, 12, 9, 9, 8, //100 x = 12,14
  8, 8, 8, 12, 12, 12, 12, 8, 8, 8,  //110 x = 14,15
12, 8, 8, 8, 6, 6, 6, 12, 12, 12, 12};  //120 x = 15,16,18,19

void sse_mult()
{
  asm("_intel_syntax noprefix\n");
  asm("mov ebx, offb\n");
  asm("mov ecx, m\n");
  asm("and ecx, 1\n");
  asm("mov edx, offb\n");
  asm("add ecx, m\n");
  asm("xor esi, esi\n");
  asm("mov eax, ecx\n");
  asm("xor edi, edi\n");
  asm("shl eax, 3\n");
  asm("xorps xmm0, xmm0\n");
  asm("xorps xmm1, xmm1\n");

  asm("cmp ecx, 6\n");
  asm("jl loop_end\n");
  asm("mult_loop:\n");
  asm("mov ebp, [l1_cache+ebx+192]\n");
  asm("movapd xmm2, [l1_cache+ebx+32]\n");
  asm("movapd xmm3, [l1_cache+ebx+32]\n");
  asm("movapd xmm4, [l1_cache+ebx+16]\n");
  asm("movapd xmm5, [l1_cache+edx+16]\n");
  asm("movapd xmm6, [l1_cache+ebx]\n");
  asm("movapd xmm7, [l1_cache+edx]\n");
  asm("mulpd xmm2, xmm5\n");
  asm("mulpd xmm3, xmm6\n");
  asm("mulpd xmm4, xmm5\n");
  asm("mulpd xmm6, xmm7\n");
  asm("haddpd xmm0, xmm2\n");
  asm("haddpd xmm1, xmm3\n");
  asm("haddpd xmm4, xmm6\n");
  asm("sub ebx, eax\n");
  asm("movapd xmm2, [l1_cache+ebx+32]\n");
  asm("movapd xmm6, [l1_cache+ebx]\n");
  asm("haddpd xmm1, xmm4\n");
  asm("movapd xmm4, [l1_cache+ebx+16]\n");
  asm("movapd xmm7, [l1_cache+edx]\n");
  asm("mulpd xmm2, xmm5\n");
  asm("mulpd xmm6, xmm7\n");
  asm("haddpd xmm0, xmm2\n");
  asm("haddpd xmm1, xmm4\n");
  asm("sub ebx, eax\n");
  asm("cmp ecx, 6\n");
  asm("jl loop_end\n");
  asm("jmp mult_loop\n");
  asm("reset_esi:\n");
  asm("xor esi, esi\n");

  asm("mov ebx, offa\n");
  asm("inc edi\n");

  asm("cmp edi, o\n");
  asm("je mult_end\n");

  asm("cmp ecx, 6\n");
  asm("jl loop_end\n");
  asm("jmp mult_loop\n");
  asm("mult_end:\n");

  int transposed = 0;
  int curr_level = 0;

inline void block_copy(double * tmp, double * tmp2, int mhalf, int extra)
{
  int stunt = m - extra;
  for(int i=0; i<mhalf; i++, tmp += stunt, tmp2 += mhalf)
  {
    memcpy(tmp2, tmp, mhalf*sizeof(double));
  }
}

inline void block_copyrb(double * tmp, double * tmp2, int mhalf, int extra)
{
  int stunt = m - extra;
  int shalf = mhalf - extra;
  for(int i=0; i<shalf; i++, tmp += stunt, tmp2 += mhalf)
  {
    memcpy(tmp2, tmp, shalf*sizeof(double));
    memset(tmp2 + shalf, 0, extra*sizeof(double));
  }
  for(int i=0; i<extra; i++)
    memset(tmp2, 0, mhalf*sizeof(double));
}

inline void block_copy_r(double *tmp, double * tmp2, int mhalf)
{
  for(int i=0; i<mhalf; i++, tmp += m, tmp2 += mhalf)
  {
    memcpy(tmp, tmp2, mhalf*sizeof(double));
  }
}

inline void block_copy_rr(double *tmp, double * tmp2, int mhalf, int extra)
{
  int stunt = mhalf - extra;
  for(int i=0; i<mhalf; i++, tmp += mhalf+stunt, tmp2 += mhalf)
  {
    memcpy(tmp, tmp2, stunt*sizeof(double));
  }
}

inline void block_copy_rb(double *tmp, double * tmp2, int mhalf, int extra)
{
  int stunt = mhalf - extra;
  for(int i=0; i<mhalf; i++, tmp += mhalf+stunt, tmp2 += mhalf)
  {
    memcpy(tmp, tmp2, mhalf*sizeof(double));
  }
}

inline void block_copy_rrb(double *tmp, double * tmp2, int mhalf, int extra)
{
  int stunt = mhalf - extra;
  for(int i=0; i<mhalf; i++, tmp += mhalf+stunt, tmp2 += mhalf)
  {
    memcpy(tmp, tmp2, stunt*sizeof(double));
  }
}

inline void block_subr(double * tmp, double * tmp2, int mhalf, int extra)
{
  int i, j;
  int stunt = mhalf - extra;
  for(i=0; i<=mhalf; i++, tmp2+=mhalf)
  {
    for(j=0; j<stunt; j++, tmp2++, tmp++)
    {
      *tmp = *tmp2 - *tmp;
    }
    for(j=0; j<extra; j++)
    {
      *tmp = -*tmp;
      tmp++;
    }
  }
}

inline void block_subb(double * tmp, double * tmp2, int mhalf, int extra)
{
  int i, j;
  int stunt = mhalf - extra;
  for(i=0; i<stunt; i++, tmp2+=stunt)
  {
    for(j=0; j<mhalf; j++, tmp2++, tmp++)
    {
      *tmp = *tmp2 - *tmp;
    }
  }
  for(i=0; i<extra; i++)
  {
    for(j=0; j<mhalf; j++, tmp++)
    {
      *tmp = -*tmp;
    }
  }
}

inline void block_addr(double * tmp, double * tmp2, int mhalf, int extra)
{
  int i, j;
  int stunt = mhalf - extra;
  for(i=0; i<mhalf; i++, tmp2+=mhalf)
  {
    for(j=0; j<stunt; j++, tmp++)
    {
      *tmp = *tmp2 + *tmp;
    }
  }
}

inline void block_addb(double * tmp, double * tmp2, int mhalf, int extra)
{
  int i, j;
  int stunt = mhalf - extra;
  for(i=0; i<mhalf; i++, tmp2+=mhalf)
  {
    for(j=0; j<stunt; j++, tmp++)
    {
      *tmp = *tmp2 + *tmp;
      tmp2++;
    }
    for(j=0; j<extra; j++, tmp++)
    {
      *tmp = *tmp2;
      tmp2++;
    }
  }
}

inline void block_add_rb(double * tmp, double * tmp2, int mhalf, int extra)
{
  int i, j;
  int stunt = mhalf - extra;
  for(i=0; i<stunt; i++, tmp2+=stunt)
  {
    for(j=0; j<mhalf; j++, tmp2++, tmp++)
    {
      *tmp2 += *tmp;
    }
  }
}

inline void block_add_rrb(double * tmp, double * tmp2, int mhalf, int extra)
{
  int i, j;
  int stunt = mhalf - extra;
  for(i=0; i<stunt; i++, tmp2+=mhalf)
  {
    for(j=0; j<mhalf; j++, tmp2++, tmp++)
    {
      *tmp2 += *tmp;
    }
    for(j=0; j<extra; j++, tmp++)
    {
      *tmp2 += *tmp;
      tmp2++;
    }
  }
}

inline void block_sub_r(double * tmp, double * tmp2, int mhalf)
{
  int i, j;
  for(i=0; i<mhalf; i++, tmp2+=mhalf)
  {
    for(j=0; j<mhalf; j++, tmp2++, tmp++)
    {
      *tmp2 -= *tmp;
    }
  }
}

inline void block_sub_rb(double * tmp, double * tmp2, int mhalf, int extra)
{
  int i, j;
  int stunt = mhalf - extra;
  for(i=0; i<stunt; i++, tmp2+=mhalf)
  {
    for(j=0; j<mhalf; j++, tmp2++, tmp++)
    {
      *tmp2 -= *tmp;
    }
    for(j=0; j<extra; j++, tmp++)
    {
      *tmp2 -= *tmp;
      tmp2++;
    }
  }
}

inline void block_sub_rrb(double * tmp, double * tmp2, int mhalf, int extra)
{
  int i, j;
  int stunt = mhalf - extra;
  for(i=0; i<mhalf; i++, tmp2+=stunt)
  {
    for(j=0; j<mhalf; j++, tmp2++, tmp++)
    {
      *tmp2 -= *tmp;
    }
    for(j=0; j<extra; j++, tmp++)
    {
      *tmp2 -= *tmp;
      tmp2++;
    }
  }
}

inline void block_add_cr(double * tmp, double * tmp2, int mhalfsq)
{
  for(int i=0; i<mhalfsq; i++, tmp++, tmp2++)
  {
    *tmp2 += *tmp;
  }
}

inline void block_sub(double * tmp, double * tmp2, int mhalf)
{
  int i, j;
  for(i=0; i<mhalf; i++, tmp2+=mhalf)
  {
    for(j=0; j<mhalf; j++, tmp2++, tmp++)
    {
      *tmp = *tmp2 - *tmp;
    }
  }
}

void
square_dgemm (const int M,
const double *A, const double *B, double *C)
{
  double * a = (double *) B;
  double * b = (double *) A;

  double * b_ptr;
  int modd = M&1;
    int i, j;

  double
  *m0, *m1, *m2, *m3, *m4, *m5, *m6,
  *a0, *a1, *a2, *a3, *a4, *a5, *a6,
  *b0, *b1, *b2, *b3, *b4, *b5, *b6;

  if(M < 132)          //L1 Tiling
  {
    m = M;
    int left_x = M;
    int left_y = M;
    int dir = -1;
    int size = block_size[M+modd];
    int size_x = size, size_y;
    offb = 0;          //1st argument is always at offset 0
    offa = ((size*(M+modd))<<3);    //2nd argument is always at this offset
    offc = (offa<<1);

    while(left_x > 0)
    {
      dir *= -1;       //start with dir = 1
      int width;

      size_y = size;
      left_y = M;
      if(left_x < size)
      {
        size_x = left_x;
      }

      int progress_x = M - left_x;

      double * start = l1_ptr + size*(M+modd);

      for(i=0; i<size_x; i++)      //transpose b. Since this is more expensive
      {      //than straight memcpy we choose this as the outer
        b_ptr = (double *) b + i + progress_x;    //loop.
        for(j=0; j<M; j++, start++, b_ptr+=M)
        {
          *start = *b_ptr;
        }
        if(modd)
        {
          *start = 0;
          start++;
        }
      }

      n = (size_x+1)&(~1);

      while(left_y > 0)
      {
        double * c_ptr;
        if(dir == 1)
        {
          c_ptr = C + progress_x + M*(M - left_y);
        }
        else
        {
          if(left_y == M)
          {
            size_y = (M+size_y-1)%size_y+1;
          }
          else
          {
            size_y = size;
          }
          c_ptr = C + progress_x + M*(left_y - size_y);
        }
        if(left_y < size_y)
        {
          size_y = left_y;
        }

        int a_width = M + modd;
        int offset = 0;
        int offset2 =0;

        if(dir == -1)
          a -= M*size_y;
        if(left_y < M || left_x == M)
        {
          for(j=0; j<size_y; j++)
          {
            memcpy((l1_ptr + offset, a + offset2, (M<<3));
            offset += a_width;
            offset2 += M;
          }
          if(dir == 1)
          a += M*size_y;

          left_y -= size_y;
          o = size_y;

          sse_mult();        //Do the multiplication!

          offc = (offa<<1);

          start = l1_ptr + (offc>>3);

          width = (size_x<<3);
          for(i=0;i<o;i++, c_ptr+=M, start+=n)
          {
            memcpy(c_ptr, start, width);
          }
          left_x -= size_x;
        }
        else
        {
          //transpose b
          m = M;

          int debugon = 0;

          int extra8 = M&7;
          int extra4 = M&3;
          int extra;

          if(M >= 509)
          {
            extra = 8-extra8;
            if(extra == 8) extra = 0;
            m0 = (double *) m0_0;
            m1 = (double *) m1_0;
            m2 = (double *) m2_0;
            m3 = (double *) m3_0;
            m4 = (double *) m4_0;
            m5 = (double *) m5_0;
            m6 = (double *) m6_0;
            a0 = (double *) a0_0;
            a1 = (double *) a1_0;
            a2 = (double *) a2_0;
            a3 = (double *) a3_0;
            a4 = (double *) a4_0;
            a5 = (double *) a5_0;
            a6 = (double *) a6_0;
            b0 = (double *) b0_0;
            b1 = (double *) b1_0;
            b2 = (double *) b2_0;
            b3 = (double *) b3_0;
            b4 = (double *) b4_0;
            b5 = (double *) b5_0;
            b6 = (double *) b6_0;
          }
          else if(M >= 255)
          {
            extra = 4-extra4;
            if(extra == 4) extra = 0;
            m0 = (double *) m0_1;
            m1 = (double *) m1_1;
            m2 = (double *) m2_1;
            m3 = (double *) m3_1;
            m4 = (double *) m4_1;
            m5 = (double *) m5_1;
            m6 = (double *) m6_1;
            a0 = (double *) a0_1;
            a1 = (double *) a1_1;
            a2 = (double *) a2_1;
            a3 = (double *) a3_1;
            a4 = (double *) a4_1;
            a5 = (double *) a5_1;
            a6 = (double *) a6_1;
            b0 = (double *) b0_1;
            b1 = (double *) b1_1;
            b2 = (double *) b2_1;
            b3 = (double *) b3_1;
            b4 = (double *) b4_1;
            b5 = (double *) b5_1;
            b6 = (double *) b6_1;
          }
          else
          {
            extra = 4-extra4;
            if(extra == 4) extra = 0;
            m0 = (double *) m0_2;
            m1 = (double *) m1_2;
            m2 = (double *) m2_2;
            m3 = (double *) m3_2;
            m4 = (double *) m4_2;
            m5 = (double *) m5_2;
            m6 = (double *) m6_2;
            a0 = (double *) a0_2;
            a1 = (double *) a1_2;
            a2 = (double *) a2_2;
            a3 = (double *) a3_2;
            a4 = (double *) a4_2;
            a5 = (double *) a5_2;
            a6 = (double *) a6_2;
            b0 = (double *) b0_2;
            b1 = (double *) b1_2;
            b2 = (double *) b2_2;
            b3 = (double *) b3_2;
            b4 = (double *) b4_2;
            b5 = (double *) b5_2;
            b6 = (double *) b6_2;
          }

          m = M+extra;

          int mhalf = m>>1;

          double * a11 = a;
          double * b11 = b;
          double * c11 = C;
          double * a12 = a + mhalf;
          double * b12 = b + mhalf;
          double * c12 = C + mhalf;
          double * a21 = a + mhalf*M;
          double * b21 = b + mhalf*M;
          double * c21 = C + mhalf*M;
          double * a22 = a21 + mhalf;
          double * b22 = b21 + mhalf;
          double * c22 = c21 + mhalf;

          int mhalfsq = mhalf*mhalf;

          block_copy(a11, a0, mhalf, extra);

          double * tmp = a0;
          double * tmp2 = a2;
          memcpy(tmp2, tmp, mhalfsq*sizeof(double));
          tmp2 = a4;
          memcpy(tmp2, tmp, mhalfsq*sizeof(double));
          tmp2 = a5;
          memcpy(tmp2, tmp, mhalfsq*sizeof(double));
          block_copyrb(a22, a3, mhalf, extra);
          tmp = a3;
          tmp2 = a1;
          memcpy(tmp2, tmp, mhalfsq*sizeof(double));
          tmp2 = a6;
          memcpy(tmp2, tmp, mhalfsq*sizeof(double));

          block_addsq(a3, a0, mhalfsq);
          block_addb(a1, a21, mhalf, extra);
          block_subb(a5, a21, mhalf, extra);
          block_addr(a4, a12, mhalf, extra);
          block_subr(a6, a12, mhalf, extra);

          block_copy(b11, b0, mhalf, extra);
          tmp = b0;
          tmp2 = b1;
          memcpy(tmp2, tmp, mhalfsq*sizeof(double));
          tmp2 = b3;
          memcpy(tmp2, tmp, mhalfsq*sizeof(double));

          tmp2 = b5;
          memcpy(tmp2, tmp, mhalfsq*sizeof(double));
          block_copyrb(b22, b4, mhalf, extra);

          tmp = b4;
          tmp2 = b2;
          memcpy(tmp2, tmp, mhalfsq*sizeof(double));
          tmp2 = b6;
          memcpy(tmp2, tmp, mhalfsq*sizeof(double));
          tmp2 = b0;
          block_addsq(b4, b0, mhalfsq);
          block_addr(b5, b12, mhalf, extra);
          block_subr(b2, b12, mhalf, extra);
          block_addb(b6, b21, mhalf, extra);
          block_subb(b3, b21, mhalf, extra);

          square_dgemm(mhalf, b0, a0, m0);

          square_dgemm(mhalf, b1, a1, m1);
          square_dgemm(mhalf, b2, a2, m2);
          square_dgemm(mhalf, b3, a3, m3);
          square_dgemm(mhalf, b4, a4, m4);
          square_dgemm(mhalf, b5, a5, m5);
          square_dgemm(mhalf, b6, a6, m6);

          m = M;

          if(extra)
          {
            block_copy_r(c11, m3, mhalf);
            block_copy_rb(c21, m3, mhalf, extra);
            block_copy_rr(c12, m2, mhalf, extra);
            block_copy_rrb(c22, m2, mhalf, extra);
            block_add_rrb(m0, c22, mhalf, extra);
            block_add_rr(m0, c11, mhalf, extra);
            block_sub_rex(m4, c11, mhalf, extra);
            block_add_rr(m4, c12, mhalf, extra);
            block_sub_rrb(m1, c22, mhalf);
            block_add_r(m1, c21, mhalf);
            block_add_rrb(m5, c22, mhalf, extra);
            block_add_rex(m6, c11, mhalf, extra);
          }
          else
          {
            block_copy_r(c11, m3, mhalf);
            block_copy_rb(c21, m3, mhalf, extra);
            block_copy_rr(c12, m2, mhalf, extra);
            block_copy_rrb(c22, m2, mhalf, extra);
            block_add_rrb(m0, c22, mhalf, extra);
            block_add_rr(m0, c11, mhalf, extra);
            block_sub_rex(m4, c11, mhalf, extra);
            block_add_rr(m4, c12, mhalf, extra);
            block_sub_rrb(m1, c22, mhalf);
            block_add_r(m1, c21, mhalf);
            block_add_rrb(m5, c22, mhalf, extra);
            block_add_rex(m6, c11, mhalf, extra);
          }
        }
      }
    }
  }
  else
  {
    square_dgemm(m, a, b, C);
  }
}
```

~ 850 LOC

Simplifying assumptions

- Ignored flop/mop parallelism within processor → drop arithmetic term
- Assumed randomly-accessible fast memory
- Assumed no-cost fast memory access
- Memory cost is constant, charged per word
  - Ignored cache lines / block transfers
  - Ignored bandwidth

Goals for today

- What is relationship to data movement in a parallel system?
- Learn the basics of a "real" memory hierarchy
  - Caches, TLB, memory
  - Ideas extend to disk systems
- Putting it all together:
  Algorithm design *and* engineering

- Quick follow-up on I/O optimality results
  - Irony, Toledo, & Tiskin (*J. PDC* 2004): "Communication lower bounds for distributed-memory matrix multiplication"

$$\text{per-processor I/Os} \quad \geq \quad \frac{n^3}{2\sqrt{2} \cdot p \cdot \sqrt{M}} - M$$

- *Theorem*: "Memory-communication trade-off"
  - # of I/Os depends on per-processor memory, M
  - Need $M \geq n^2 / (2 \cdot p^{2/3})$
  - What does this mean for algorithm designers? for computer architects?

$$\text{per-processor I/Os} \quad \geq \quad \frac{n^2}{4\sqrt{2 \cdot \mu p}}$$

- *Theorem*: "2-D lower bound"
  - Assuming each processor has
    - $M = \mu \cdot n^2 / p$
    - $p \geq 32\mu^3$
  - Cannon, Fox, SUMMA algorithms all match in terms of volume of communication

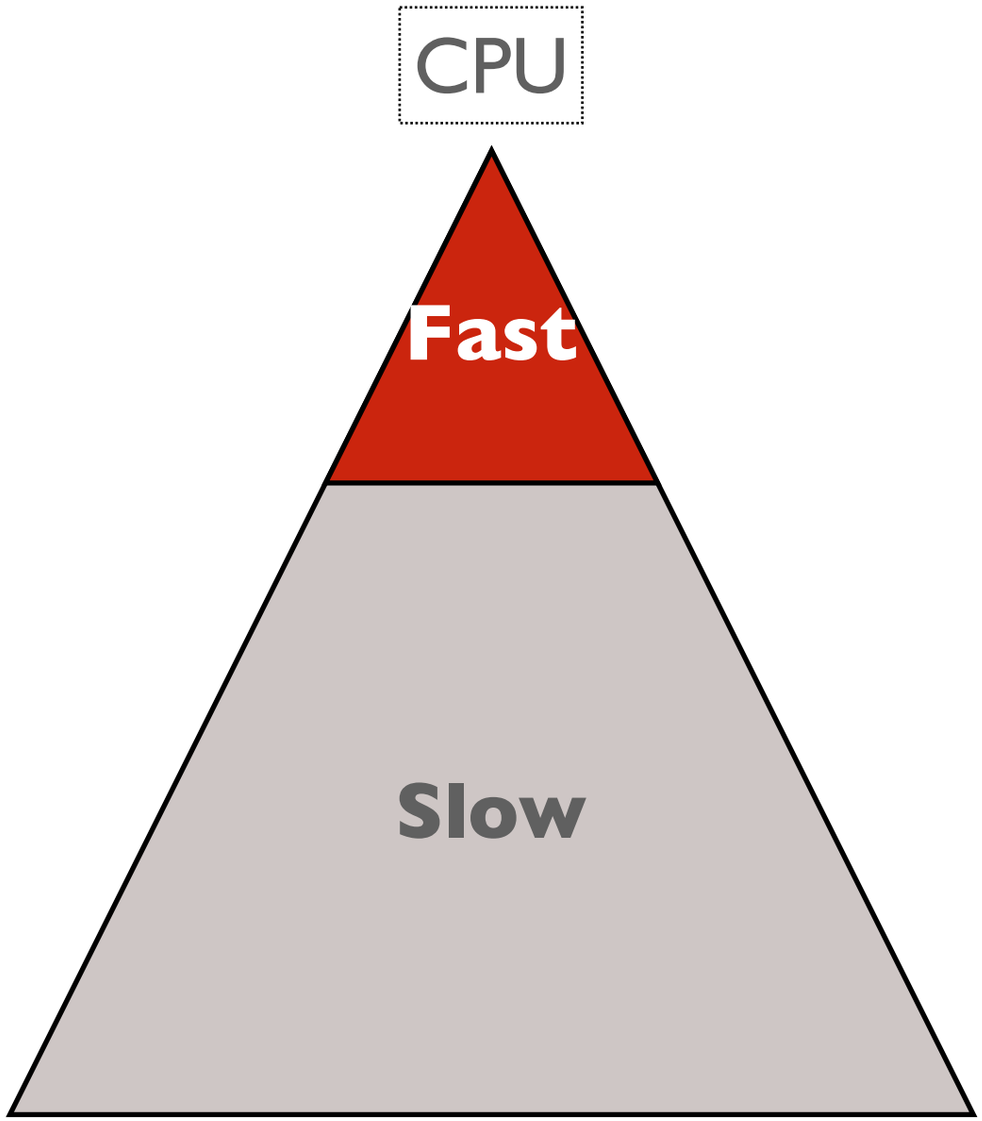
Bandwidth vs. latency

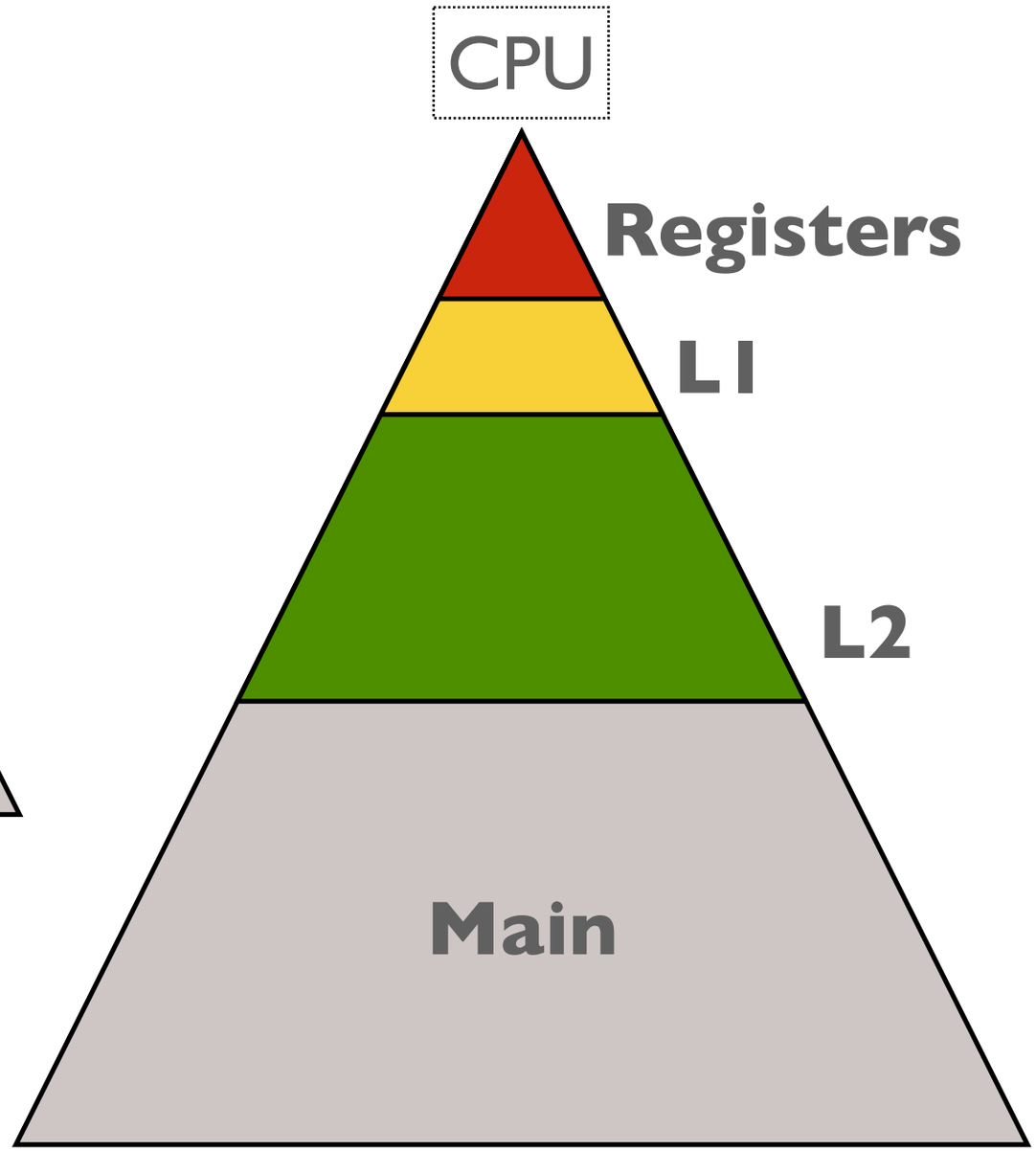
- Message cost model to send *n* words

$$T(n) = \alpha + \frac{n}{\beta} \qquad \alpha = \text{latency (time)}, \beta = \text{bandwidth (words/time)}$$

- I/O optimality results cover bandwidth term
  - For latency bound, divide our I/O results by max message size, *M*
  - Flurry of new research on latency-optimal algorithms: See Hoemmen and Demmel at UC Berkeley

- Back to memory hierarchies:
  What does a cache look like?


CPU
Fast
Slow


CPU
Registers
L1
L2
Main

Dealing with high memory latency

- Motivation for **caches** & **prefetching**
  - Data likely to be reused (**temporal locality**)
  - Likely to access adjacent data (**spatial locality**)
  - Bandwidth improves faster than latency
- H/W provides automated support
  - All loads cached automatically (LRU), and loaded in chunks (*cache line size*)
  - Typical to have a hardware prefetcher that detects simple patterns

Cache terms

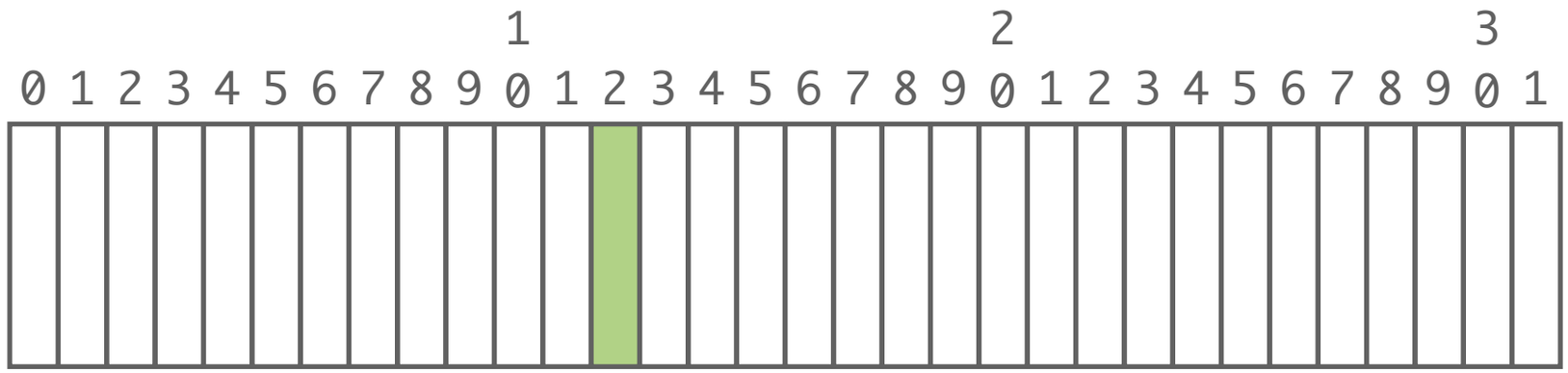
- "**Hit**" = load/store data in cache, *vs.* "**Miss**"
- Cache data transferred in **blocks** or **lines**
- Conceptual categories of misses
  - **Compulsory**: First access
  - **Capacity**: Cache cannot contain all blocks needed to execute the program
  - **Conflict**: Block replaced but then accessed soon thereafter

Cache design questions

- **Placement**: Where in cache does a block go?
  - Associativity
- **Replacement**: Replace which block on a miss?
  - Random, LRU, FIFO (round-robin)
- **Write strategy**: What happens on a write?
  - Write-through vs. write-back
  - Write-allocate or not

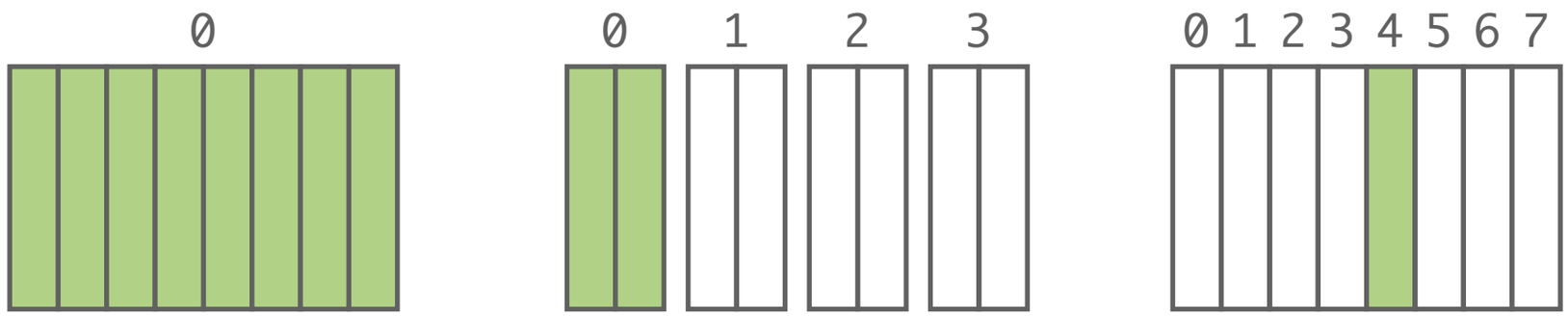
Placement

Memory



Q: Where does block #12 go?

Cache



Fully associative:
Anywhere

(2-way) set associative:
mod(12, 4) = 0

Direct mapped
mod(12, 8) = 4

Replacement

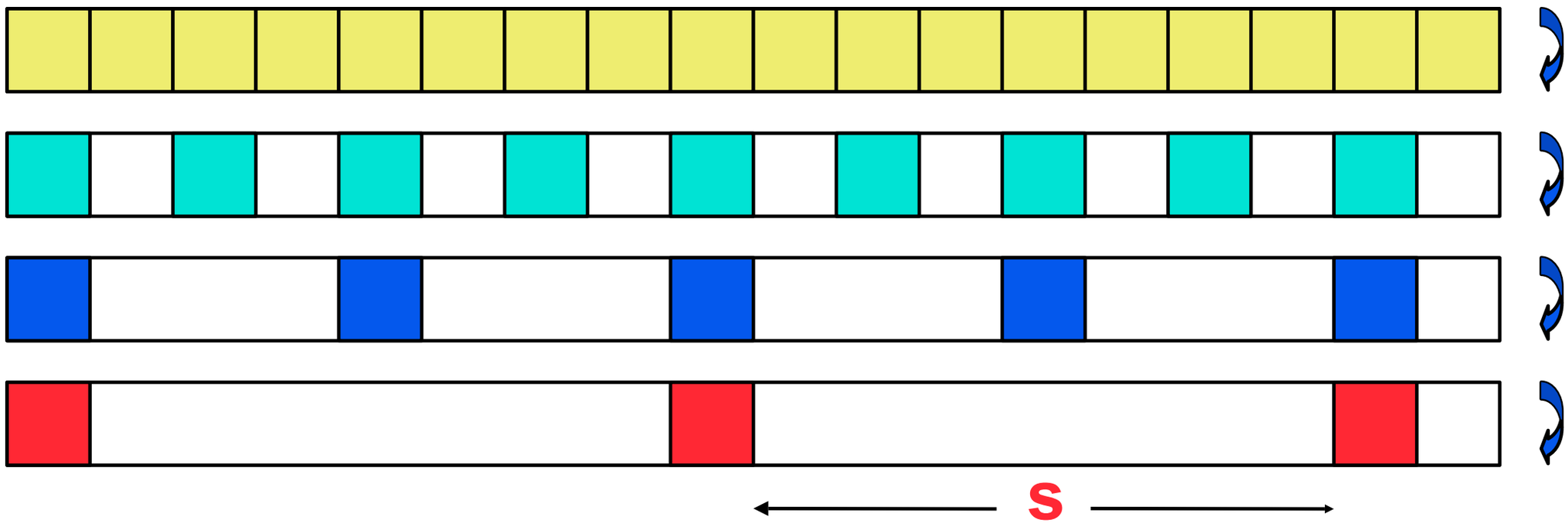
- Set- or fully-associative
  - Random: "Easy" to implement, but non-determinism makes life complicated for us
  - Least recently used (LRU): Hard to implement efficiently as associativity increases
  - Round-robin (first-in, first-out): Often used when associativity is high
- Victim cache: Hold evicted blocks
- Inclusive vs. exclusive multi-level caches

Write policies

- Cache hit
  - Write-through: ↑ traffic, but simplifies coherence
  - Write-back: Write only on eviction
- Cache miss
  - No write allocate: Write to main memory only
  - Allocate: Fetch into cache

Cache terms (refresher)

- Motivating intuition: Spatial & temporal locality
  - Cache hit vs. miss
  - 3C's: compulsory, capacity, conflict
- Cache design space
  - Line size
  - Associativity: fully, k-way, direct
  - Replacement: random, LRU, round-robin (FIFO)
  - Write policies: write-thru, write-back, allocate
  - Victim cache, inclusive vs. exclusive

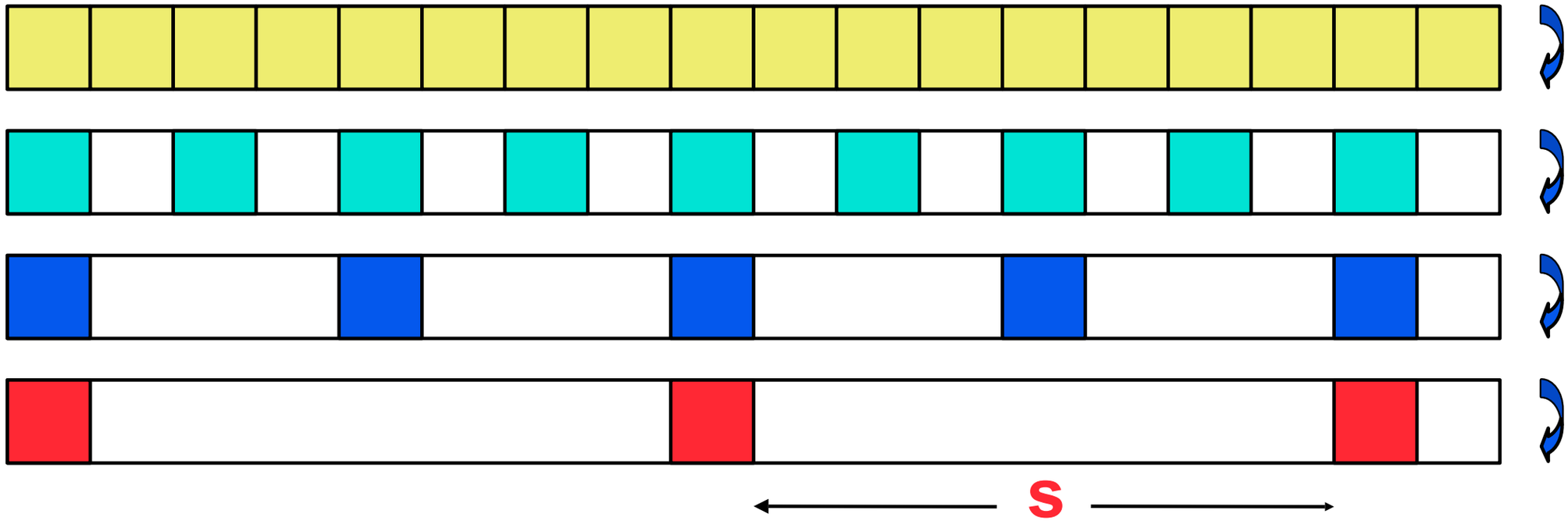



```
for L in 2, 4, 8, … do          // For each array size, L
  for s in 2, 4, 8, …, L/2 do // For each stride, s
    // Start timer
    for many trials do
      for k = 0 to L step s do  // For each element, in steps of s
        read X[k]
    // Stop timer and report average read time per element
```

But how does the M.H. affect me?

Saavedra-Barrera benchmark: Strided-stream through array, measuring average access time.

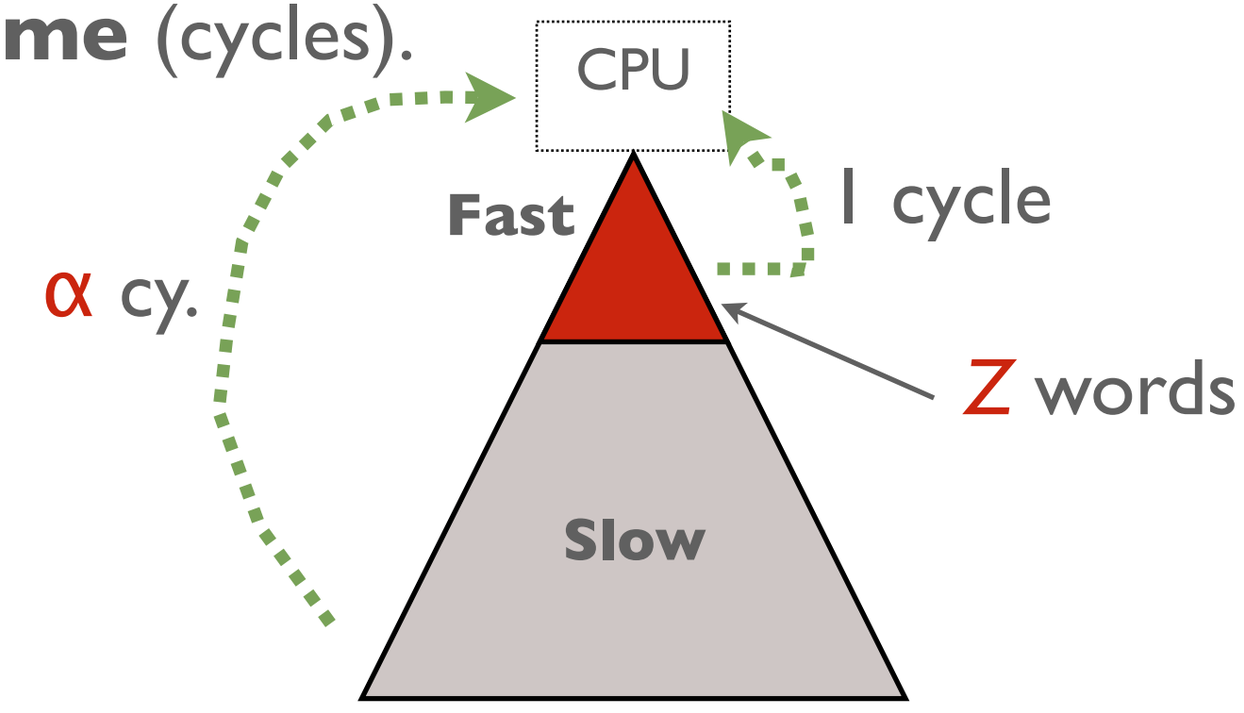
What do we expect?



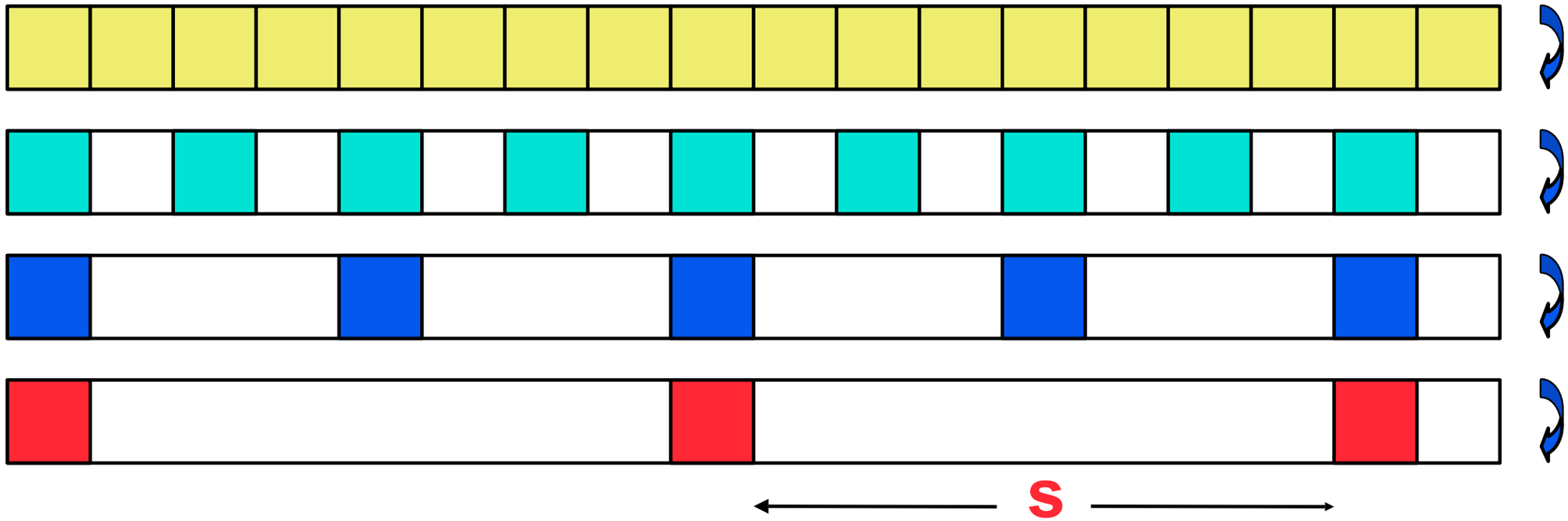
Let τ = avg. read time (cycles).

Array size: *n* words
Stride: *s* words

Line size: *l* words
Assoc.: Direct-mapped
Replacement: LRU



What do we expect?

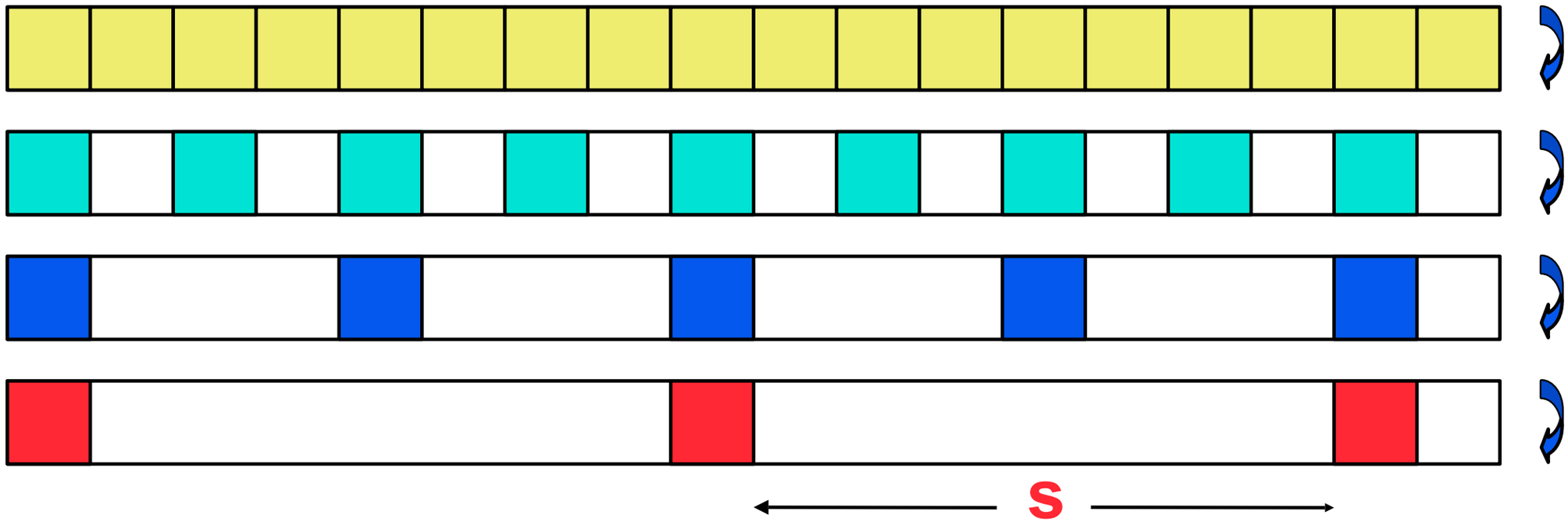


Mem. latency: α cycles
Cache size: *Z* words
Line size: *l* words
Cache latency: 1 cycle
Assoc.: Direct-mapped
Replacement: LRU
Array size: *n* words
Stride: *s* words

Let τ = avg. read time (cycles).

$$n \leq Z \quad \Longrightarrow \quad \tau = ?$$

What do we expect?



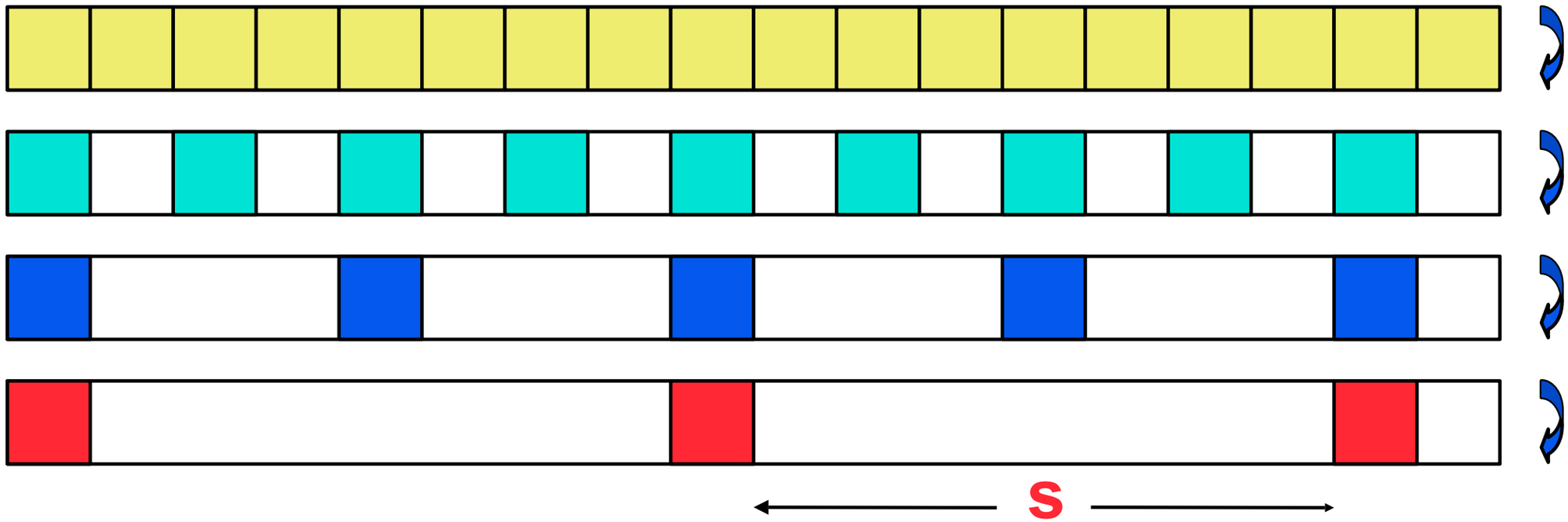
Mem. latency: α cycles
Cache size: *Z* words
Line size: *l* words
Cache latency: 1 cycle
Assoc.: Direct-mapped
Replacement: LRU
Array size: *n* words
Stride: *s* words

Let τ = avg. read time (cycles).

$$n \leq Z \quad \Longrightarrow \quad \tau = 1$$

Array cache-resident,
so τ independent of *s*.

What do we expect?



Mem. latency: α cycles
Cache size: *Z* words
Line size: *l* words
Cache latency: 1 cycle
Assoc.: Direct-mapped
Replacement: LRU
Array size: *n* words
Stride: *s* words

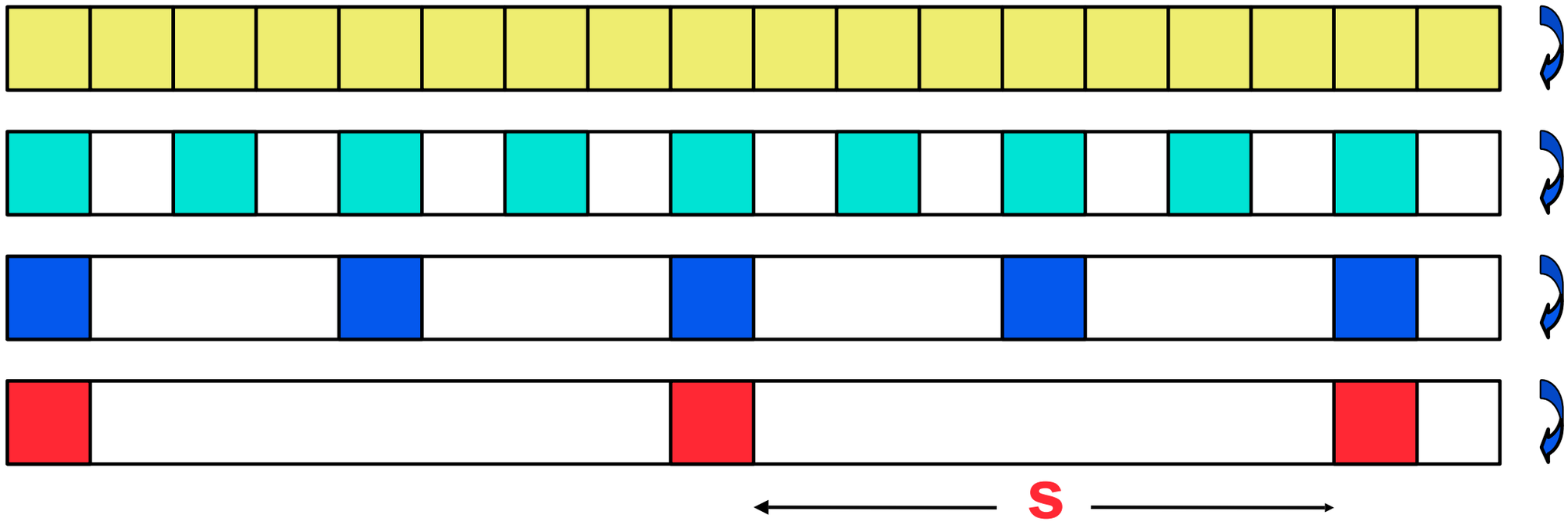
Let τ = avg. read time (cycles).

$$n \leq Z \quad \Longrightarrow \quad \tau = 1$$

$$n > Z \quad \Longrightarrow \quad \tau = ?$$

Array cache-resident,
so τ independent of *s*.

What do we expect?



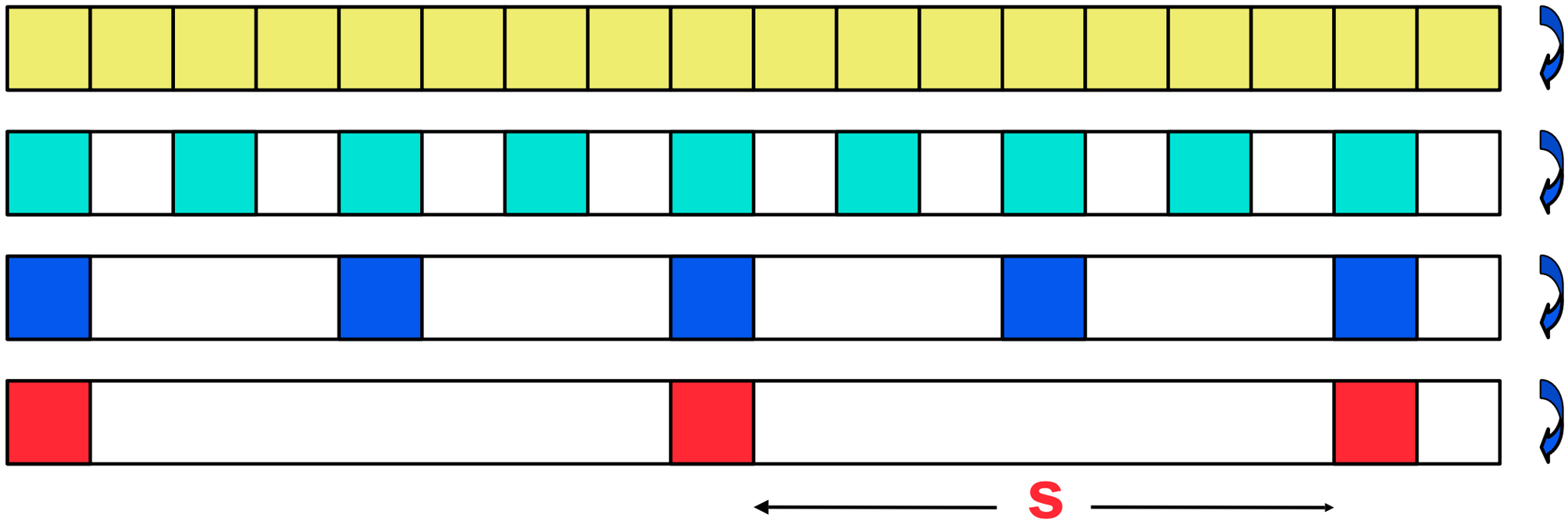
Mem. latency: α cycles
Cache size: *Z* words
Line size: *l* words
Cache latency: 1 cycle
Assoc.: Direct-mapped
Replacement: LRU
Array size: *n* words
Stride: *s* words

Let τ = avg. read time (cycles).

$$n > Z \quad \text{and} \quad s = 1$$

$$\Longrightarrow \quad \tau \quad = \quad ?$$

What do we expect?



Mem. latency: α cycles
Cache size: *Z* words
Line size: *l* words
Cache latency: 1 cycle
Assoc.: Direct-mapped
Replacement: LRU
Array size: *n* words
Stride: *s* words

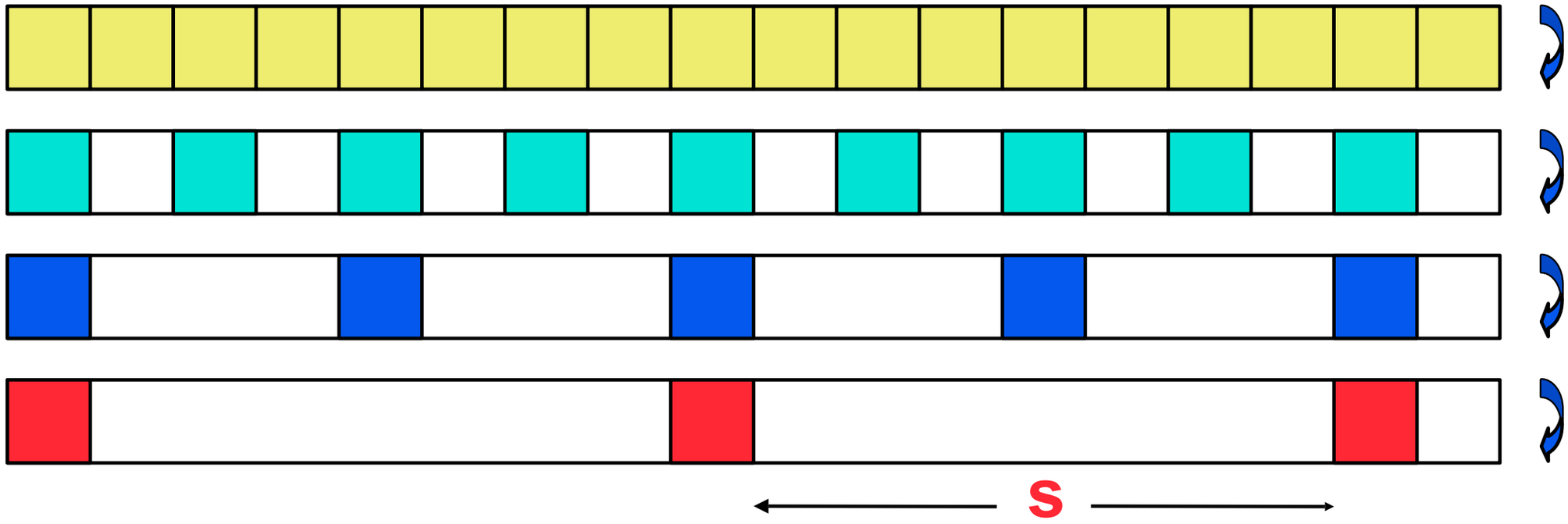
Let τ = avg. read time (cycles).

$$n > Z \quad \text{and} \quad s = 1$$

$$\Longrightarrow \quad \tau \quad = \quad \frac{\alpha + (l-1)}{l}$$

Full latency on 1st word only.

What do we expect?



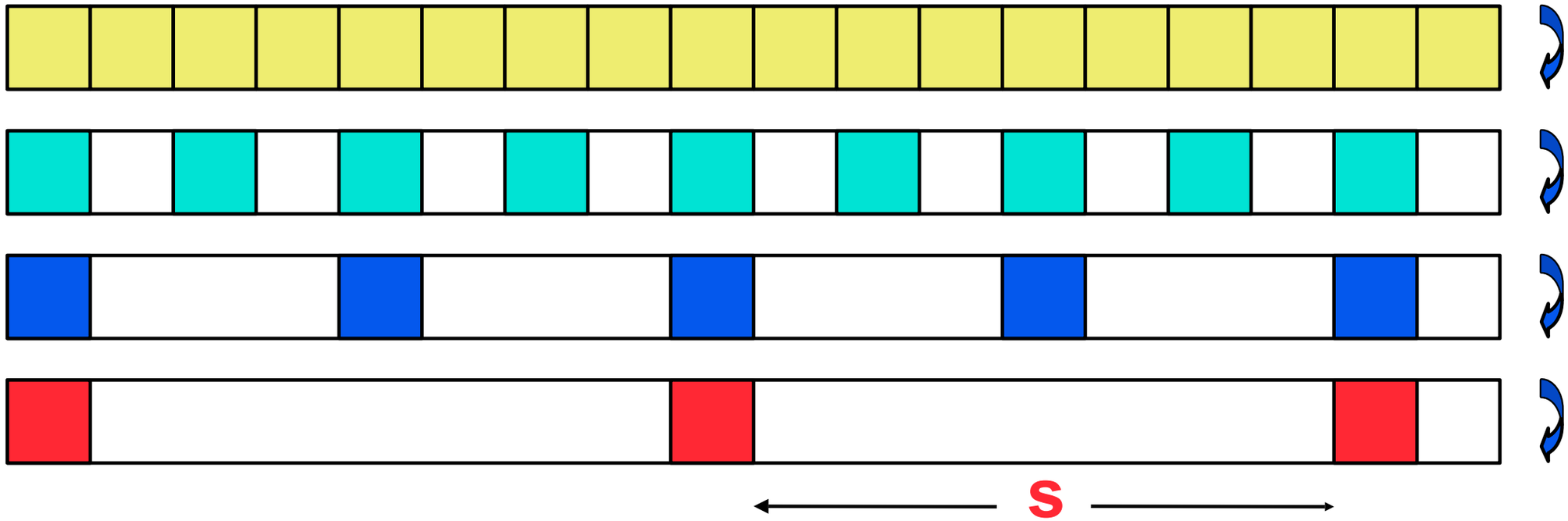
Mem. latency: α cycles
Cache size: *Z* words
Line size: *l* words
Cache latency: 1 cycle
Assoc.: Direct-mapped
Replacement: LRU
Array size: *n* words
Stride: *s* words

Let τ = avg. read time (cycles).

$$n > Z \quad \text{and} \quad s \leq l$$

$$\Longrightarrow \quad \tau \quad = \quad ?$$

What do we expect?



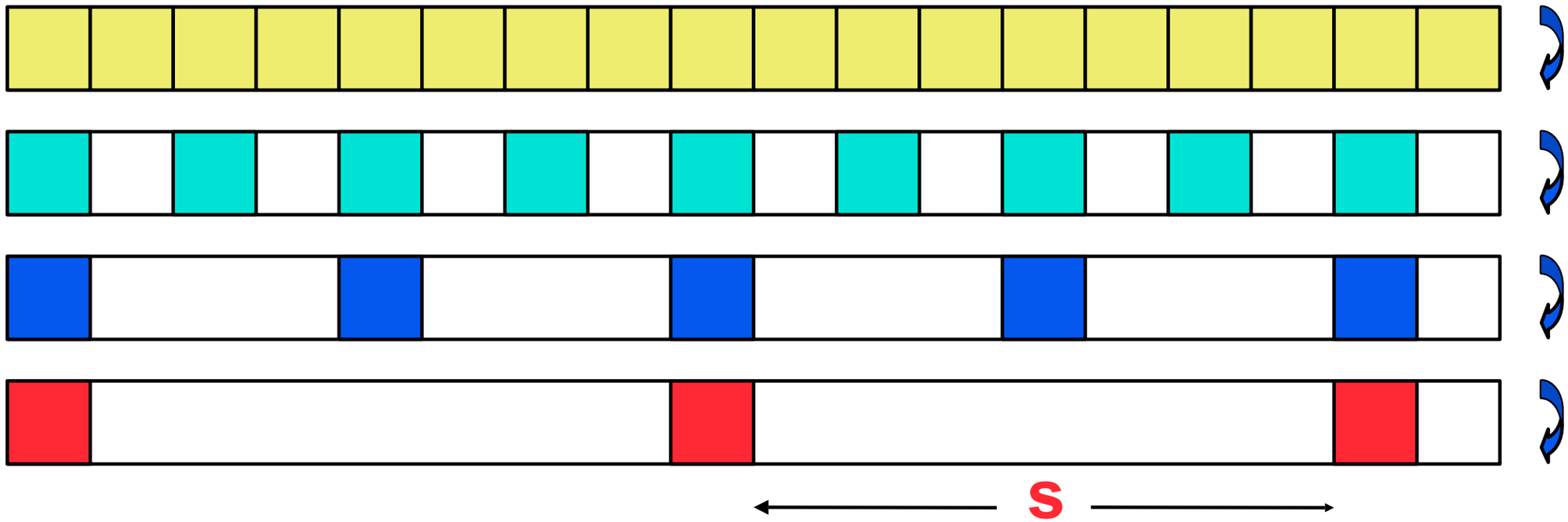
Mem. latency: α cycles
Cache size: *Z* words
Line size: *l* words
Cache latency: 1 cycle
Assoc.: Direct-mapped
Replacement: LRU
Array size: *n* words
Stride: *s* words

Let τ = avg. read time (cycles).

$$n > Z \quad \text{and} \quad s \leq l$$

$$\Longrightarrow \quad \tau \quad = \quad \frac{\alpha + \frac{l}{s} - 1}{\frac{l}{s}}$$

What do we expect?



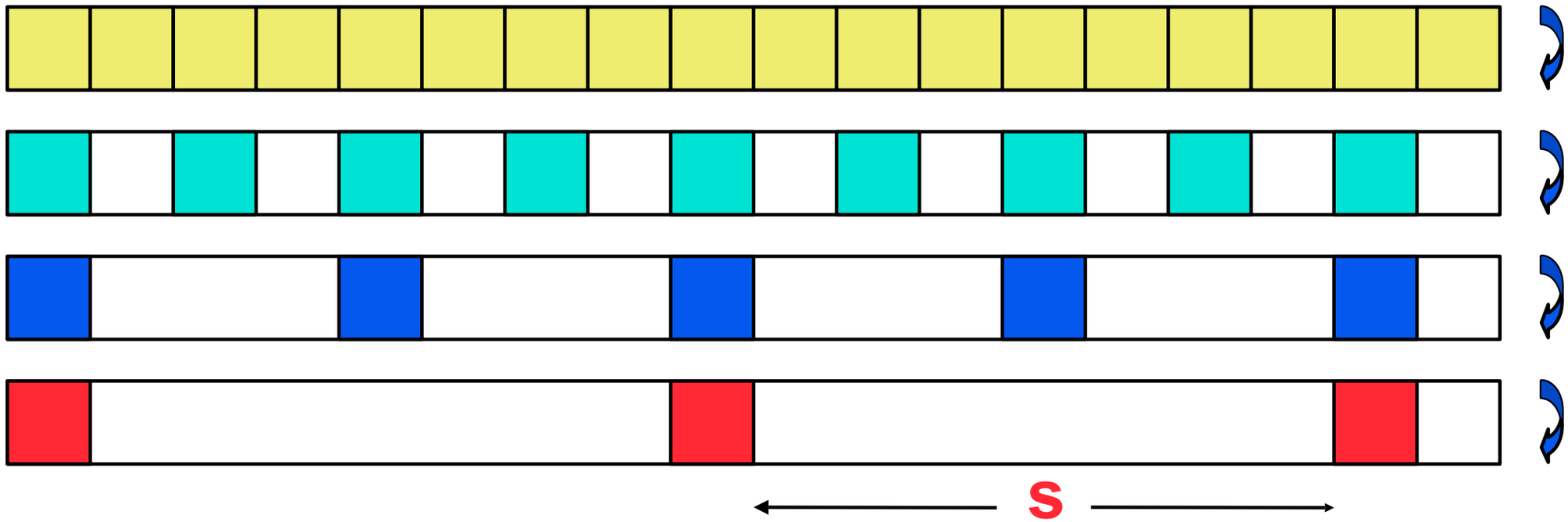
Mem. latency: α cycles
Cache size: *Z* words
Line size: *l* words
Cache latency: 1 cycle
Assoc.: Direct-mapped
Replacement: LRU
Array size: *n* words
Stride: *s* words

Let τ = avg. read time (cycles).

$$n > Z \quad \text{and} \quad s > l$$

$$\Longrightarrow \quad \tau \quad = \quad ?$$

What do we expect?



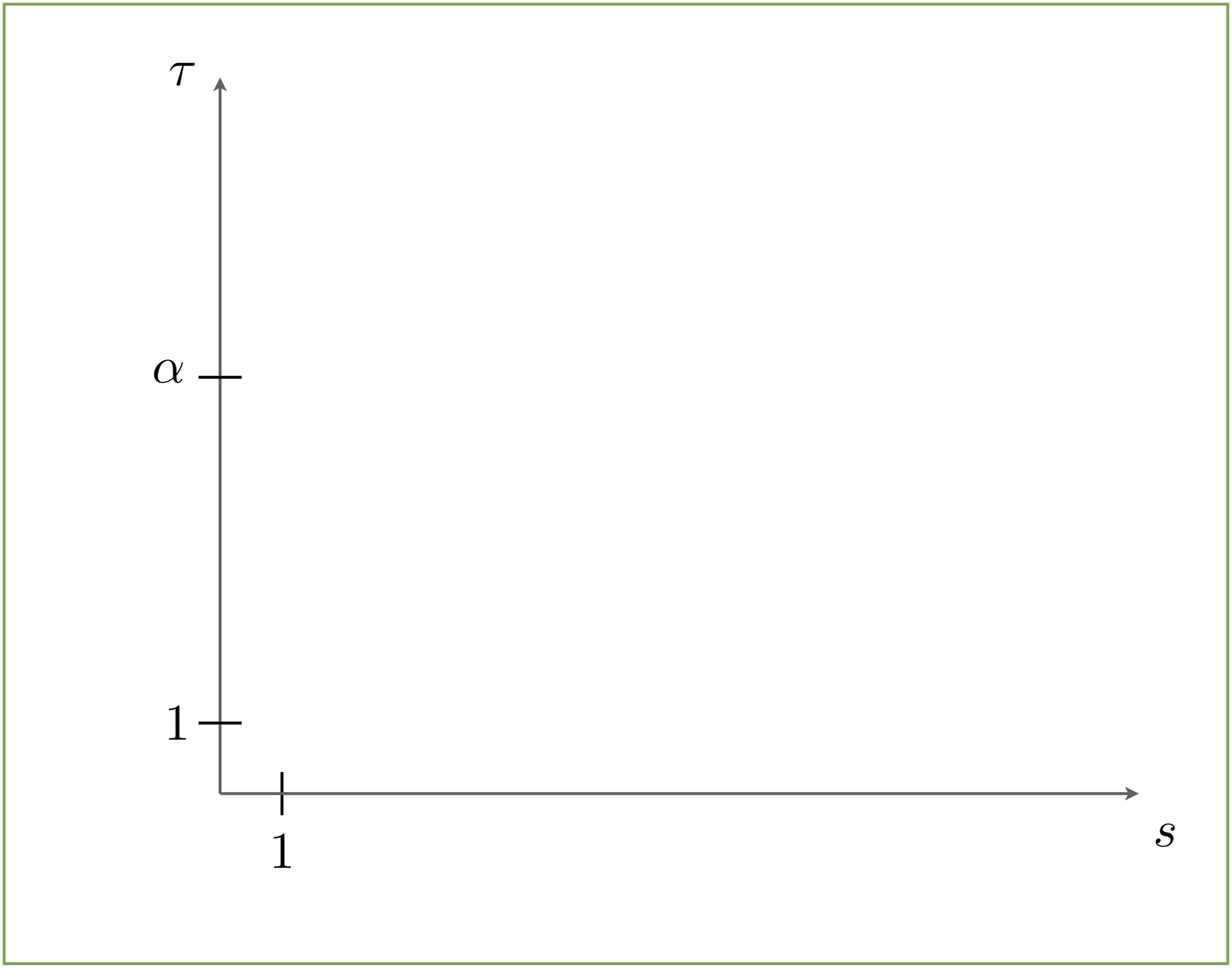
Mem. latency: α cycles
Cache size: *Z* words
Line size: *l* words
Cache latency: 1 cycle
Assoc.: Direct-mapped
Replacement: LRU
Array size: *n* words
Stride: *s* words

Let τ = avg. read time (cycles).

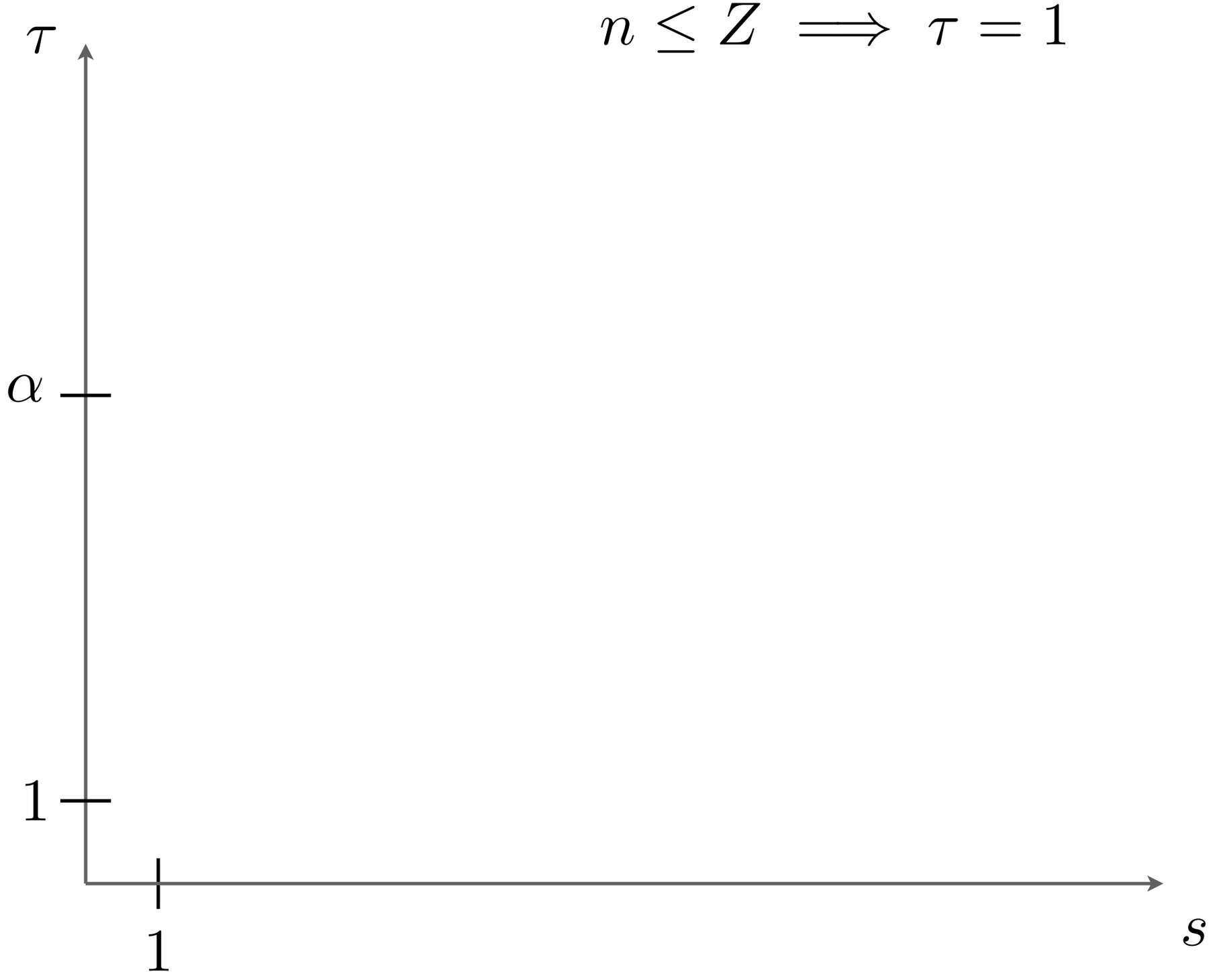
$$n > Z \quad \text{and} \quad s > l$$

$$\Longrightarrow \quad \tau \quad = \quad \alpha$$

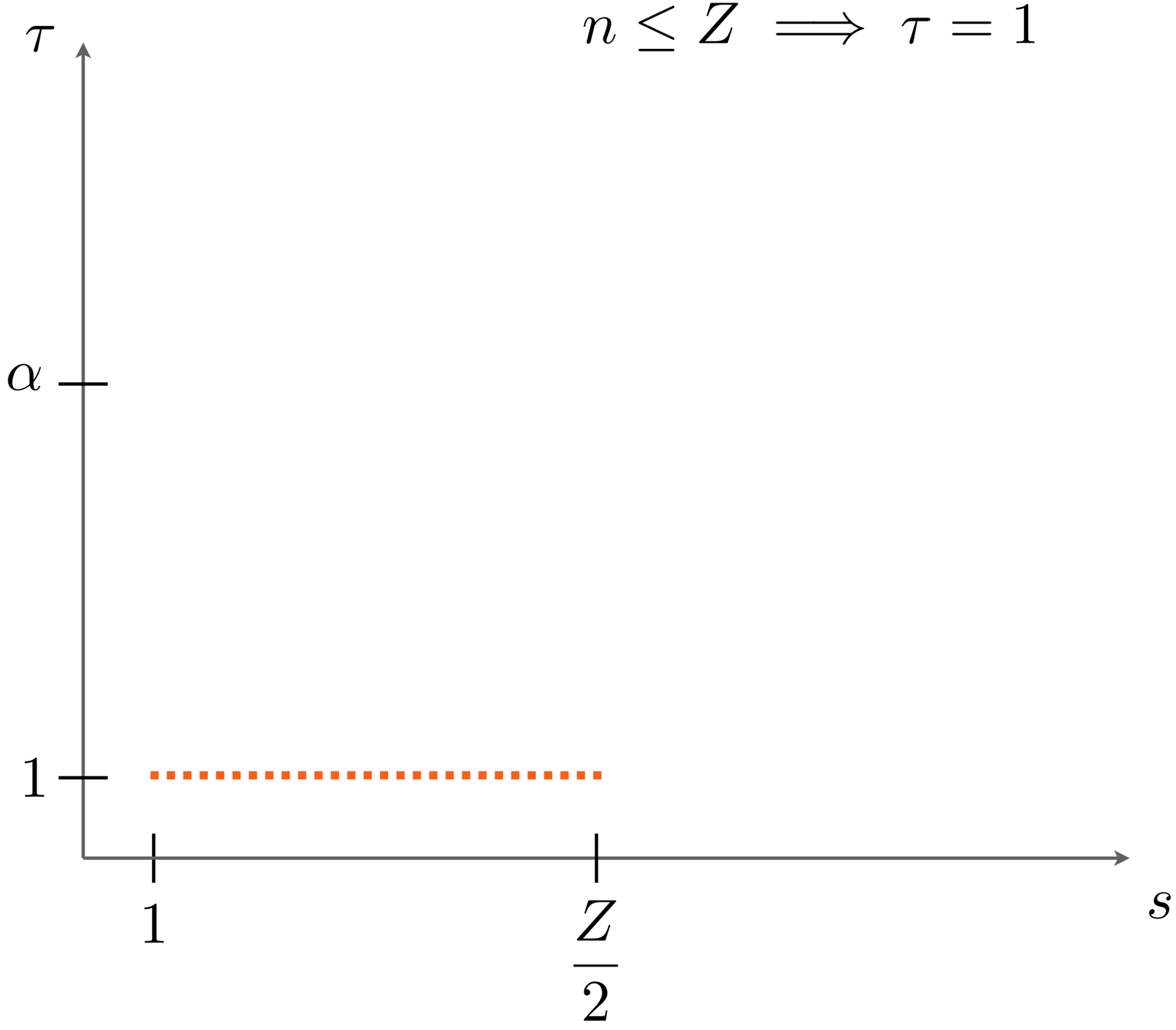
Full latency on 1st word only.


τ
α
1
1
s

$$n \leq Z \implies \tau = 1$$



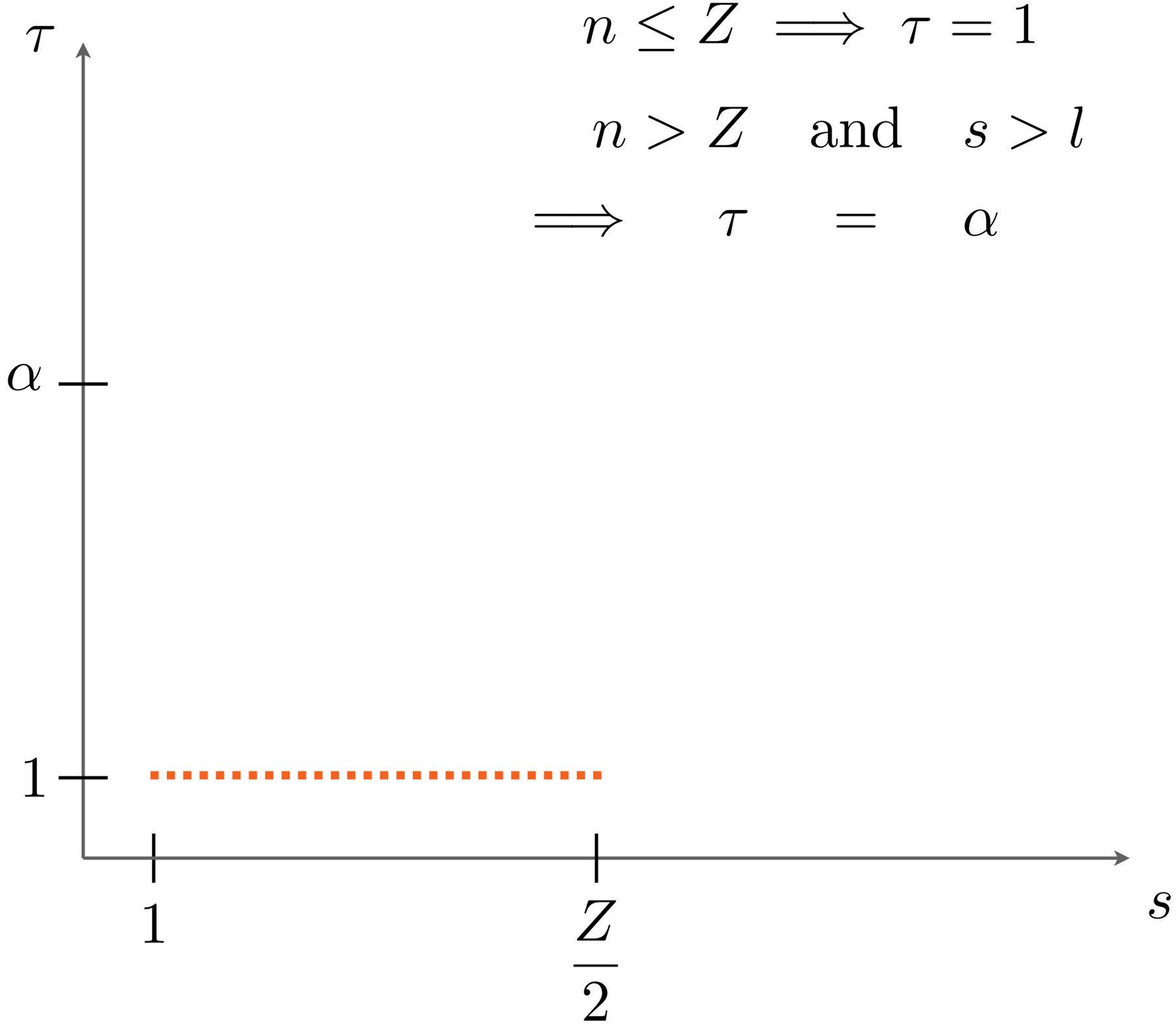
$$n \leq Z \implies \tau = 1$$



$$n \leq Z \implies \tau = 1$$

$$n > Z \quad \text{and} \quad s > l$$

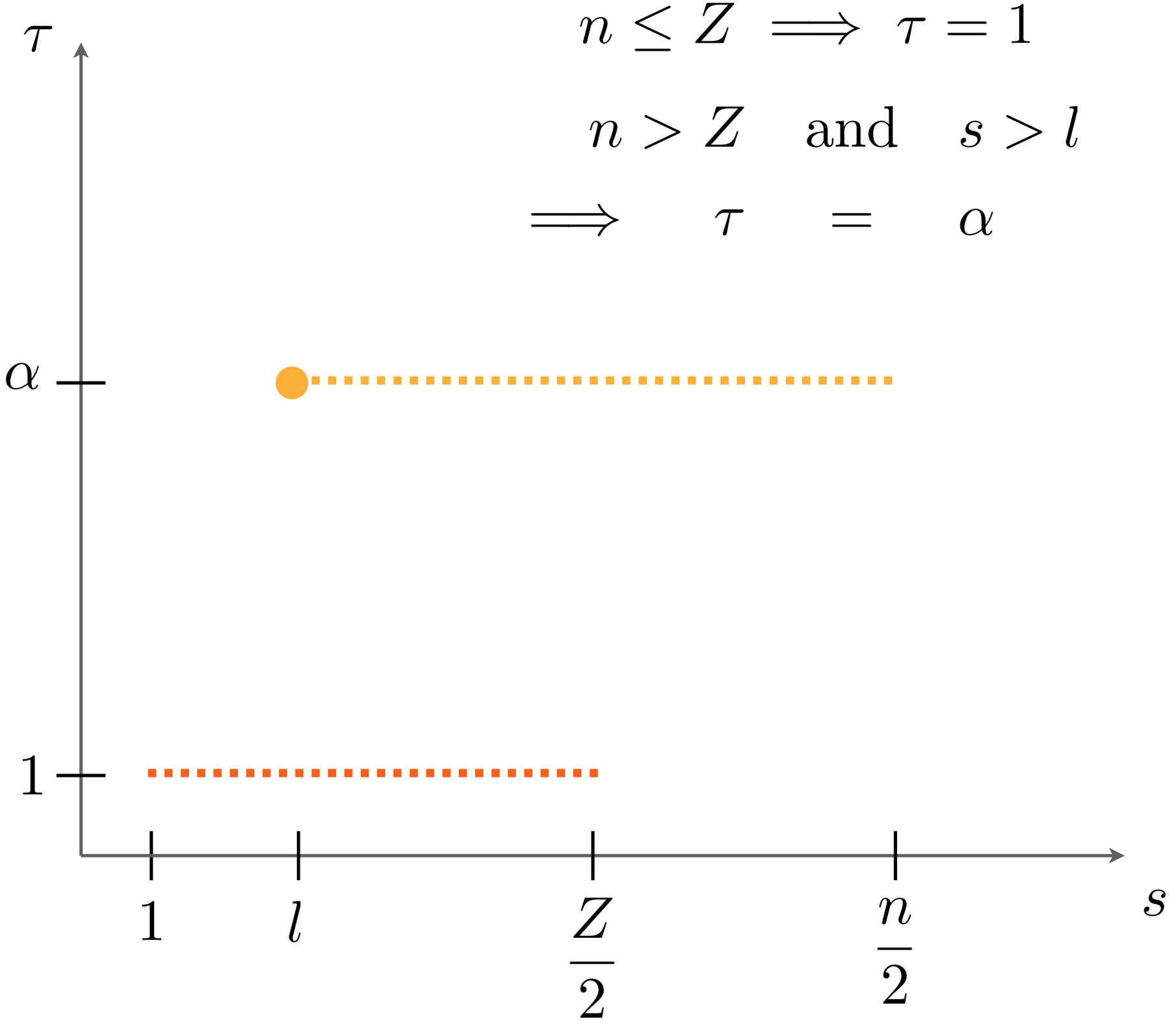
$$\implies \quad \tau \quad = \quad \alpha$$



$$n \leq Z \implies \tau = 1$$

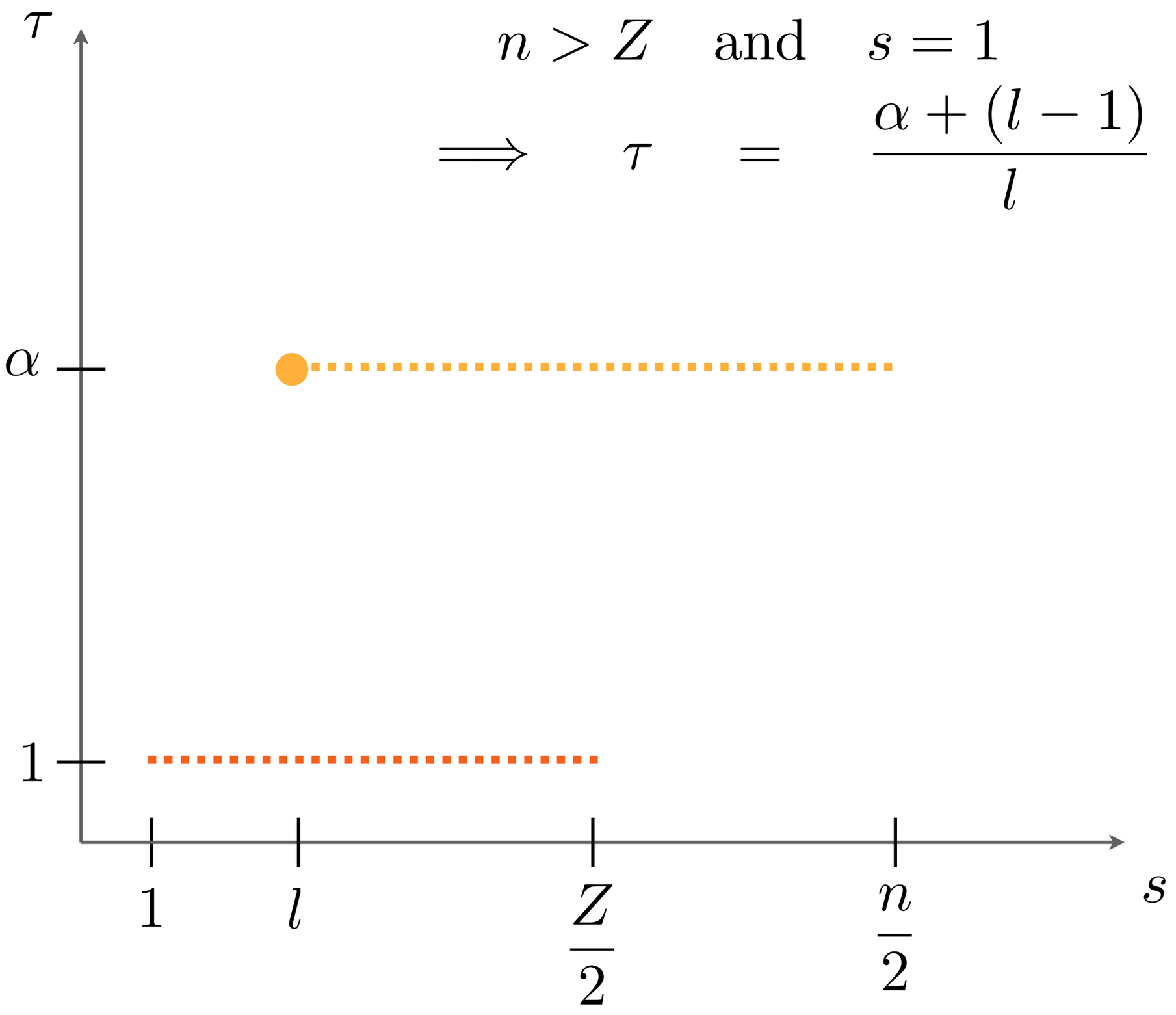
$$n > Z \quad \text{and} \quad s > l$$

$$\implies \quad \tau \quad = \quad \alpha$$



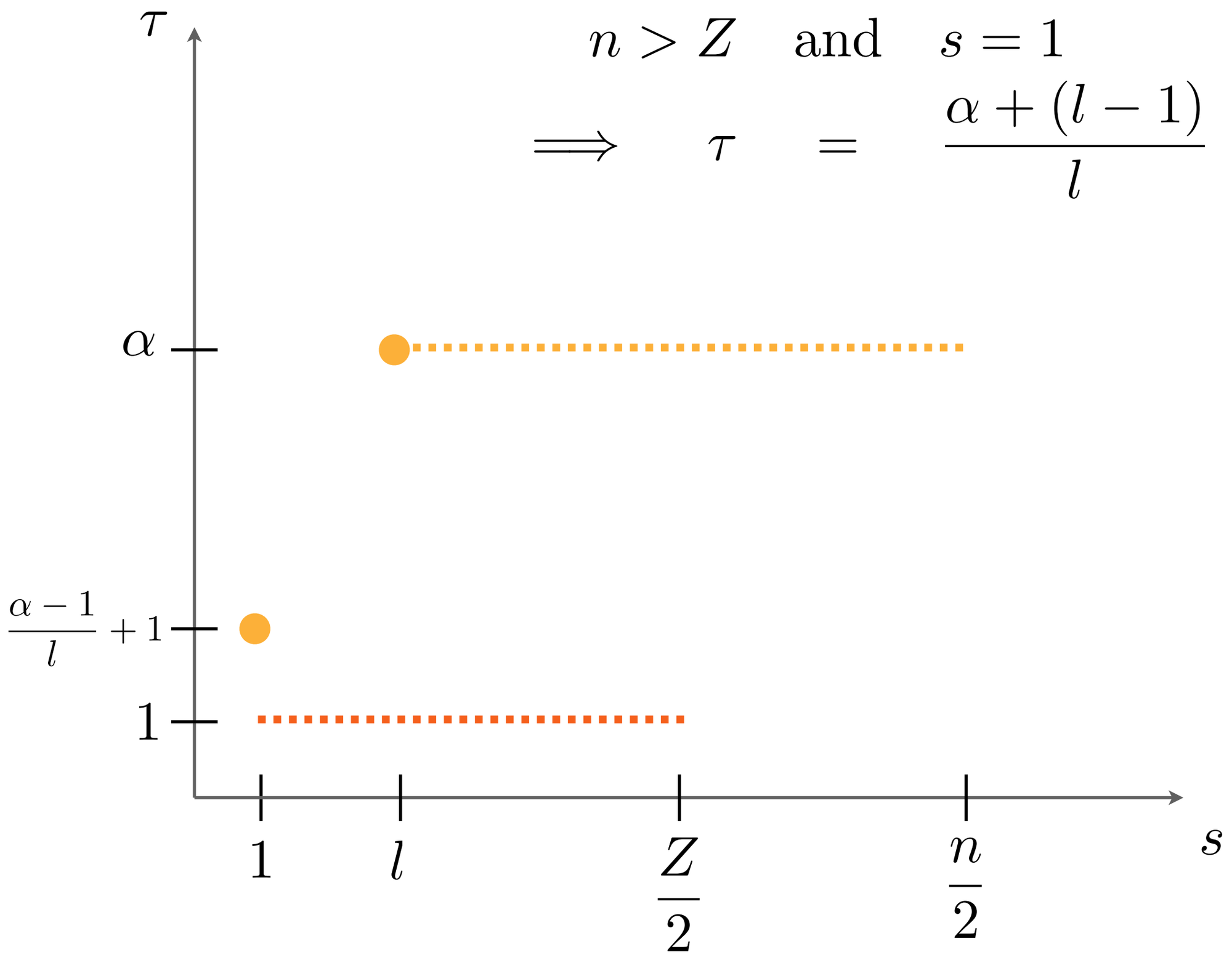
$$n > Z \quad \text{and} \quad s = 1$$

$$\Longrightarrow \quad \tau \quad = \quad \frac{\alpha + (l-1)}{l}$$



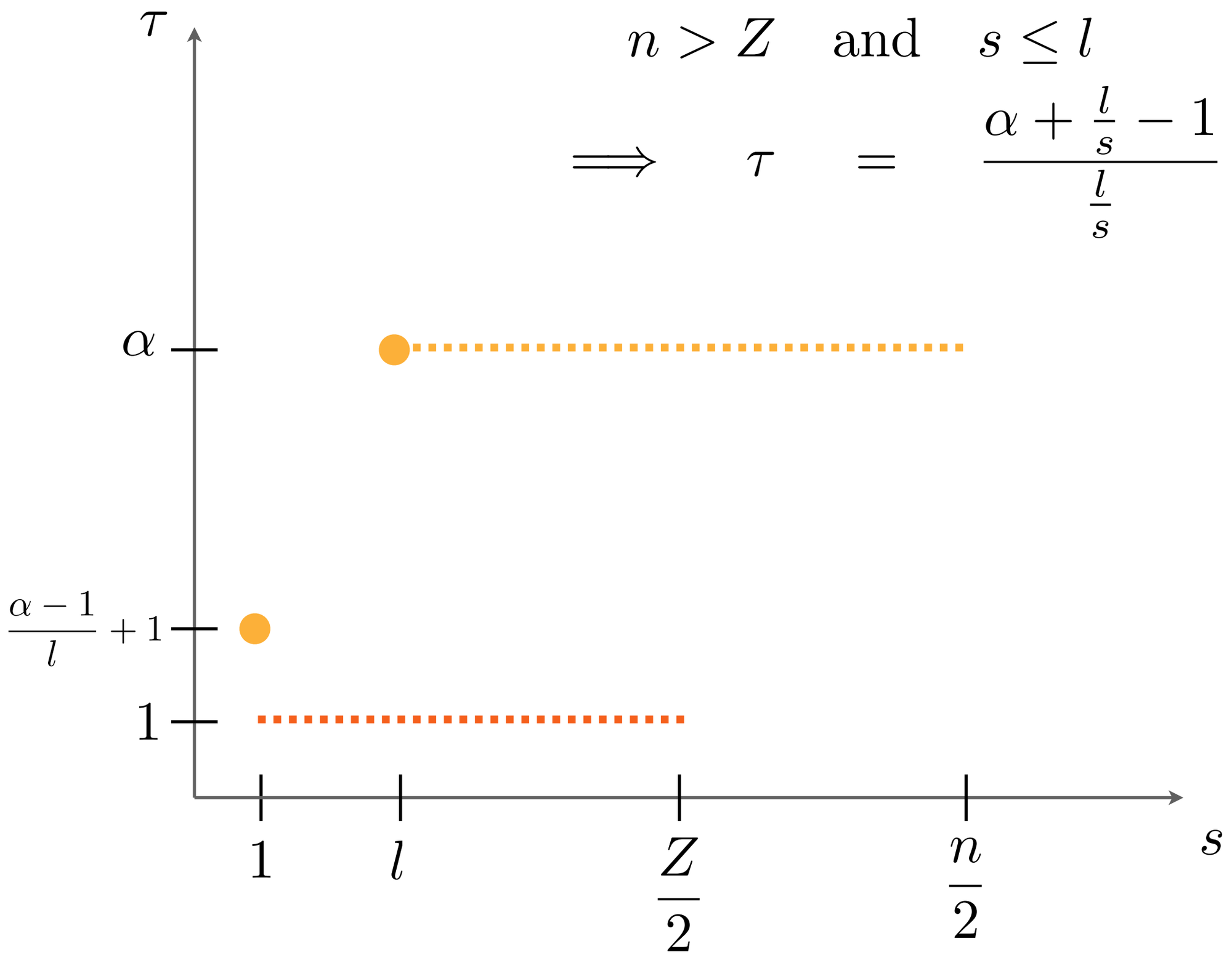
$$n > Z \quad \text{and} \quad s = 1$$

$$\Longrightarrow \quad \tau \quad = \quad \frac{\alpha + (l-1)}{l}$$



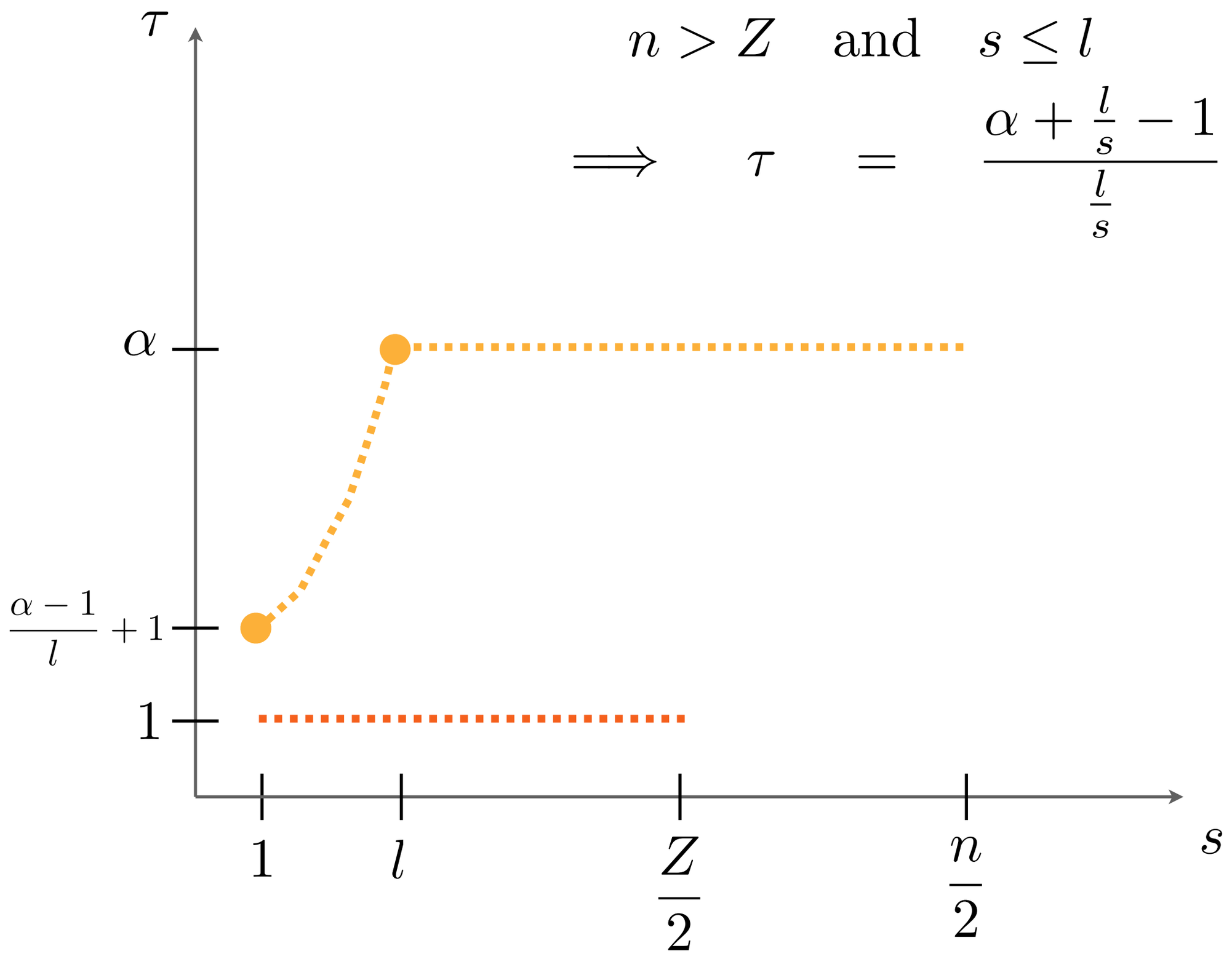
$$n > Z \quad \text{and} \quad s \leq l$$

$$\Longrightarrow \quad \tau \quad = \quad \frac{\alpha + \frac{l}{s} - 1}{\frac{l}{s}}$$

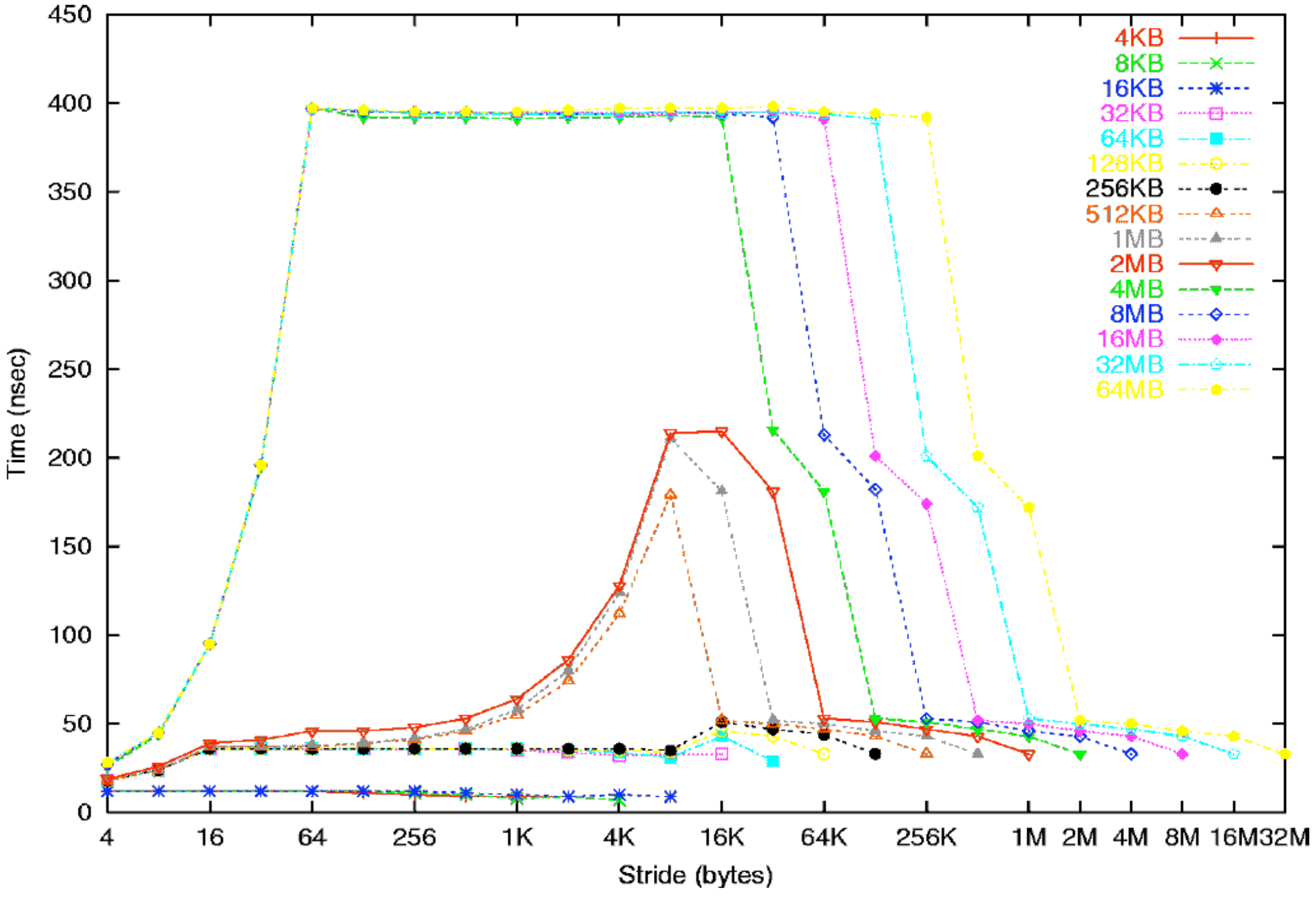


$$n > Z \quad \text{and} \quad s \leq l$$

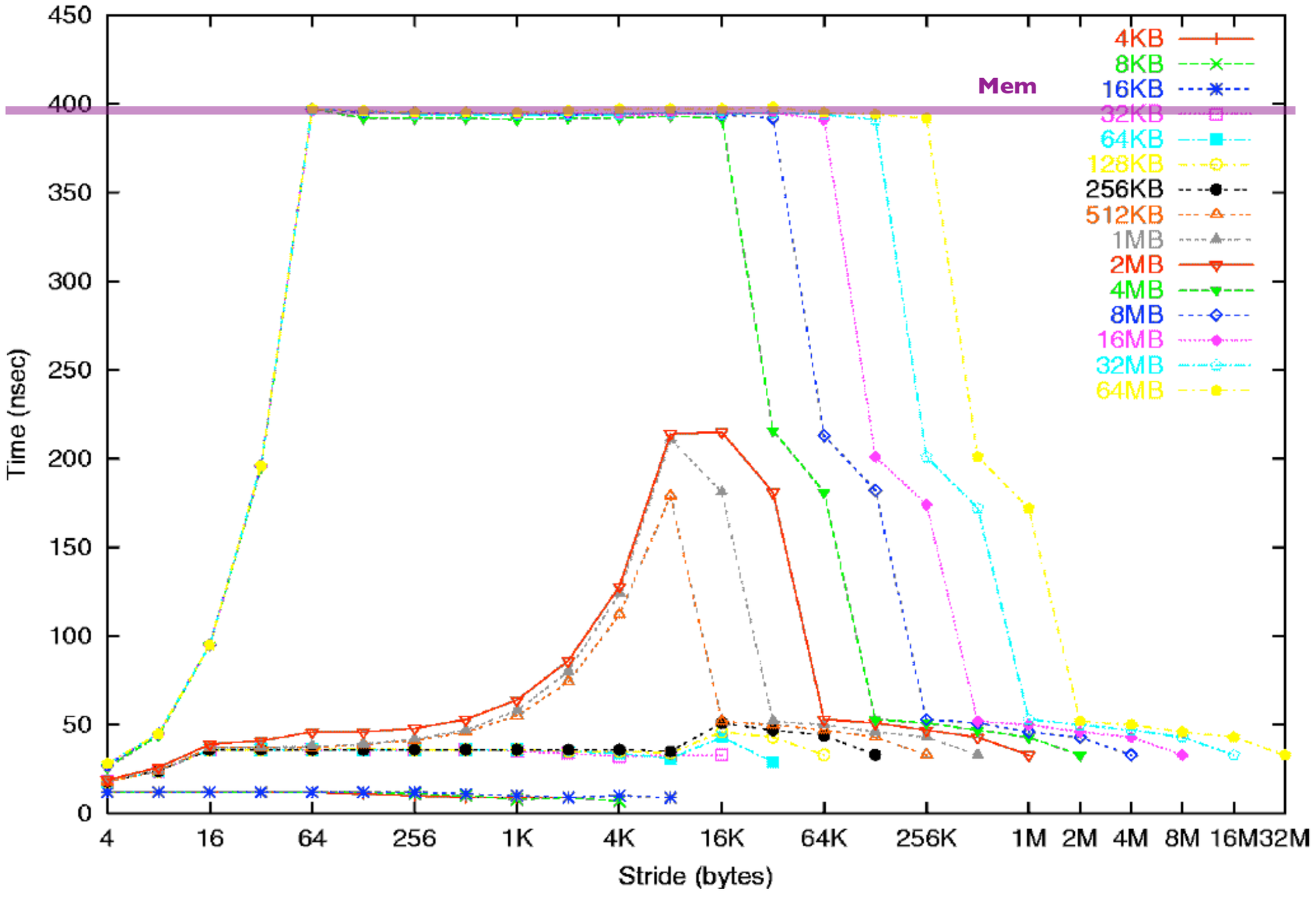
$$\Longrightarrow \quad \tau \quad = \quad \frac{\alpha + \frac{l}{s} - 1}{\frac{l}{s}}$$



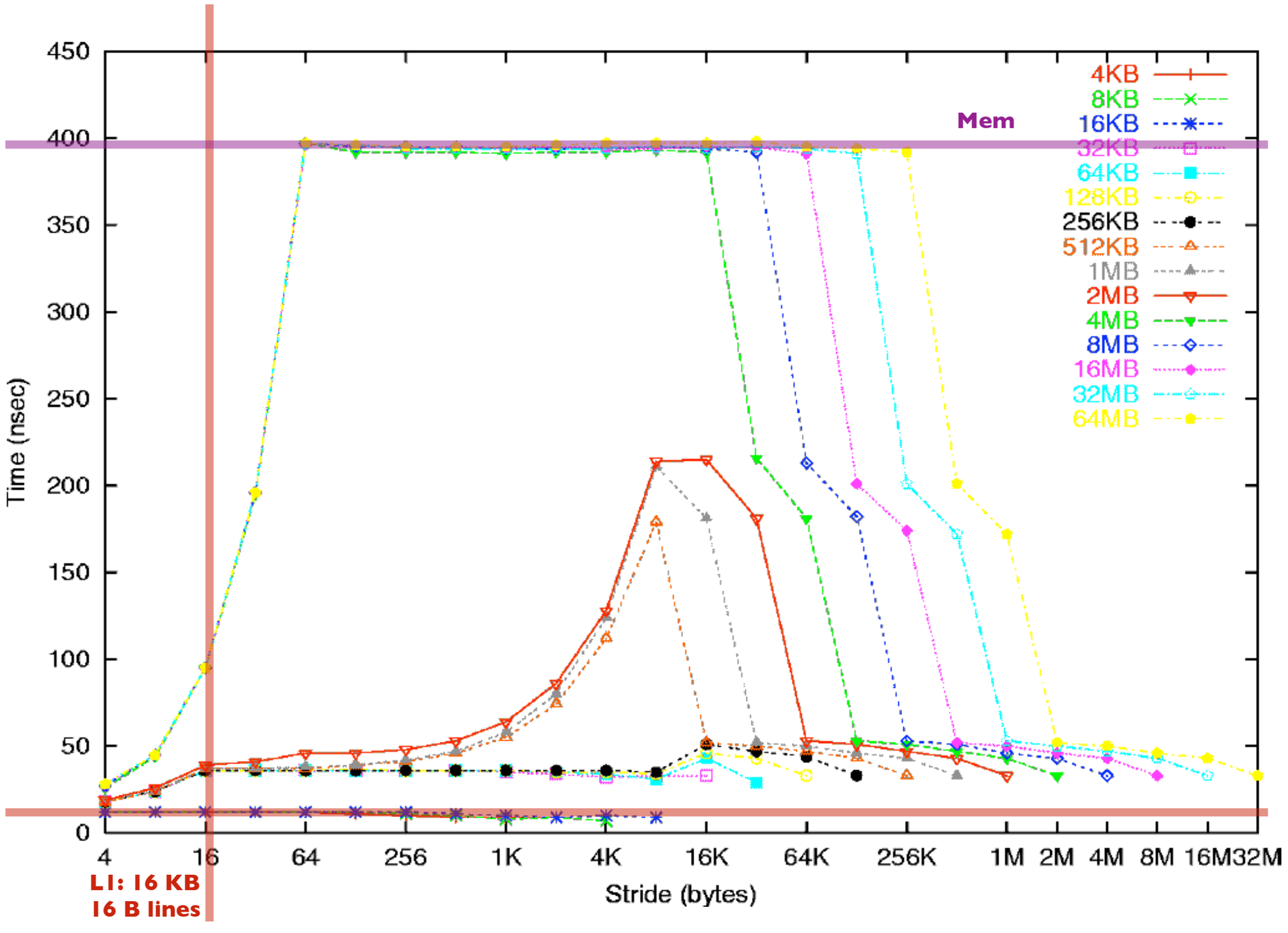
Average Memory Access Time (Saavedra-Barerra) — Sun Ultra IIi (333 MHz)



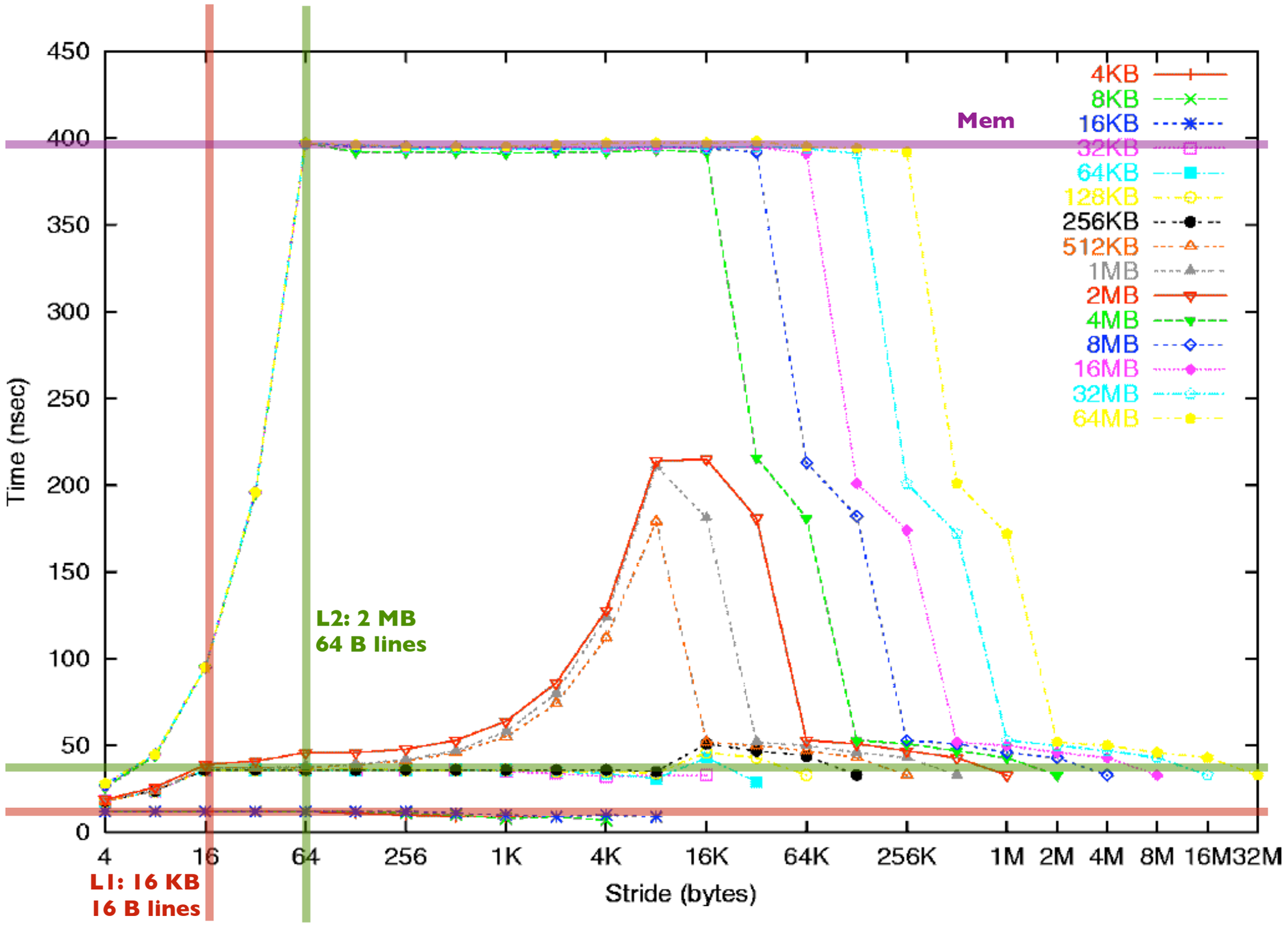
Average Memory Access Time (Saavedra-Barerra) — Sun Ultra IIi (333 MHz)



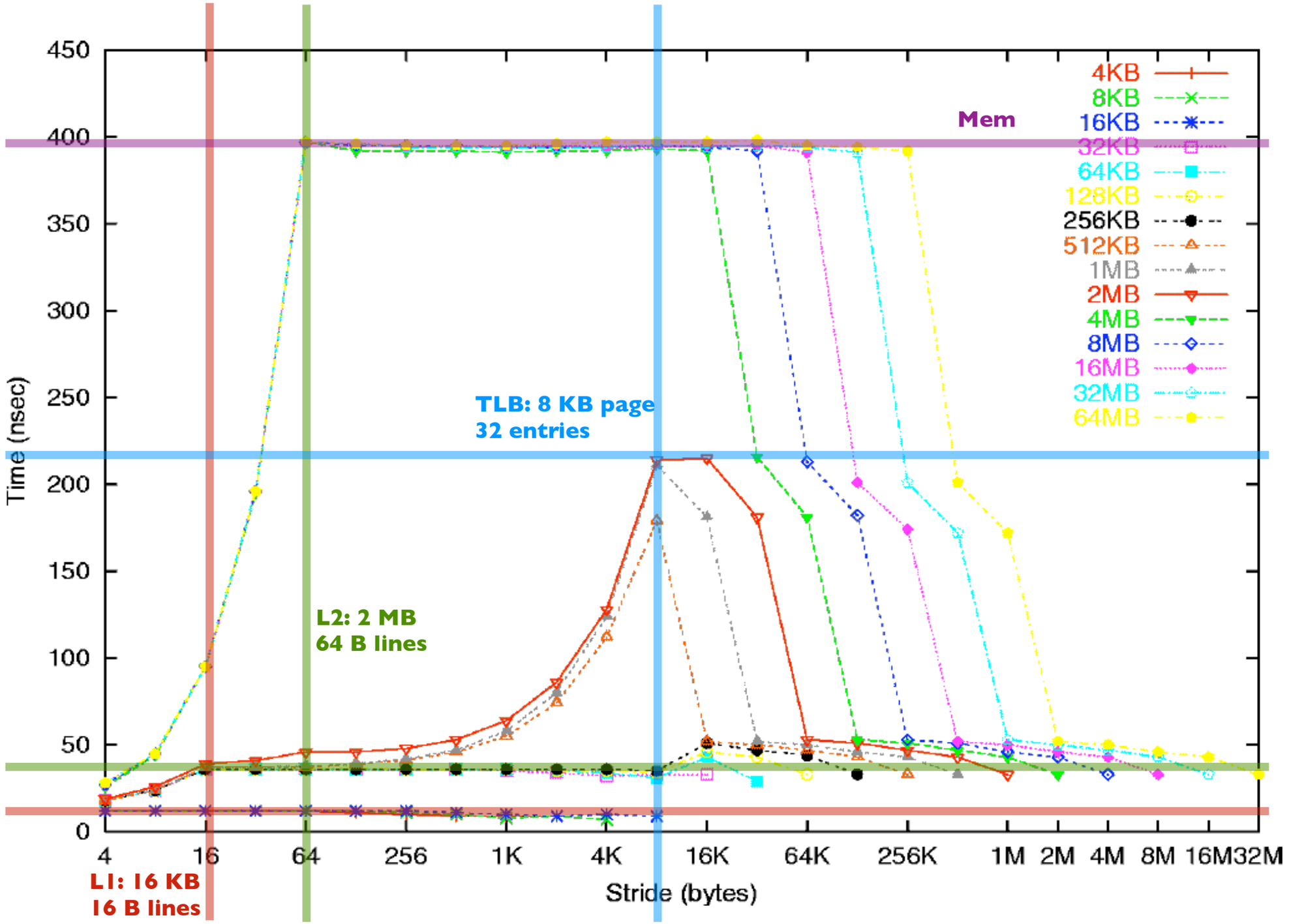
Average Memory Access Time (Saavedra-Barerra) — Sun Ultra IIi (333 MHz)



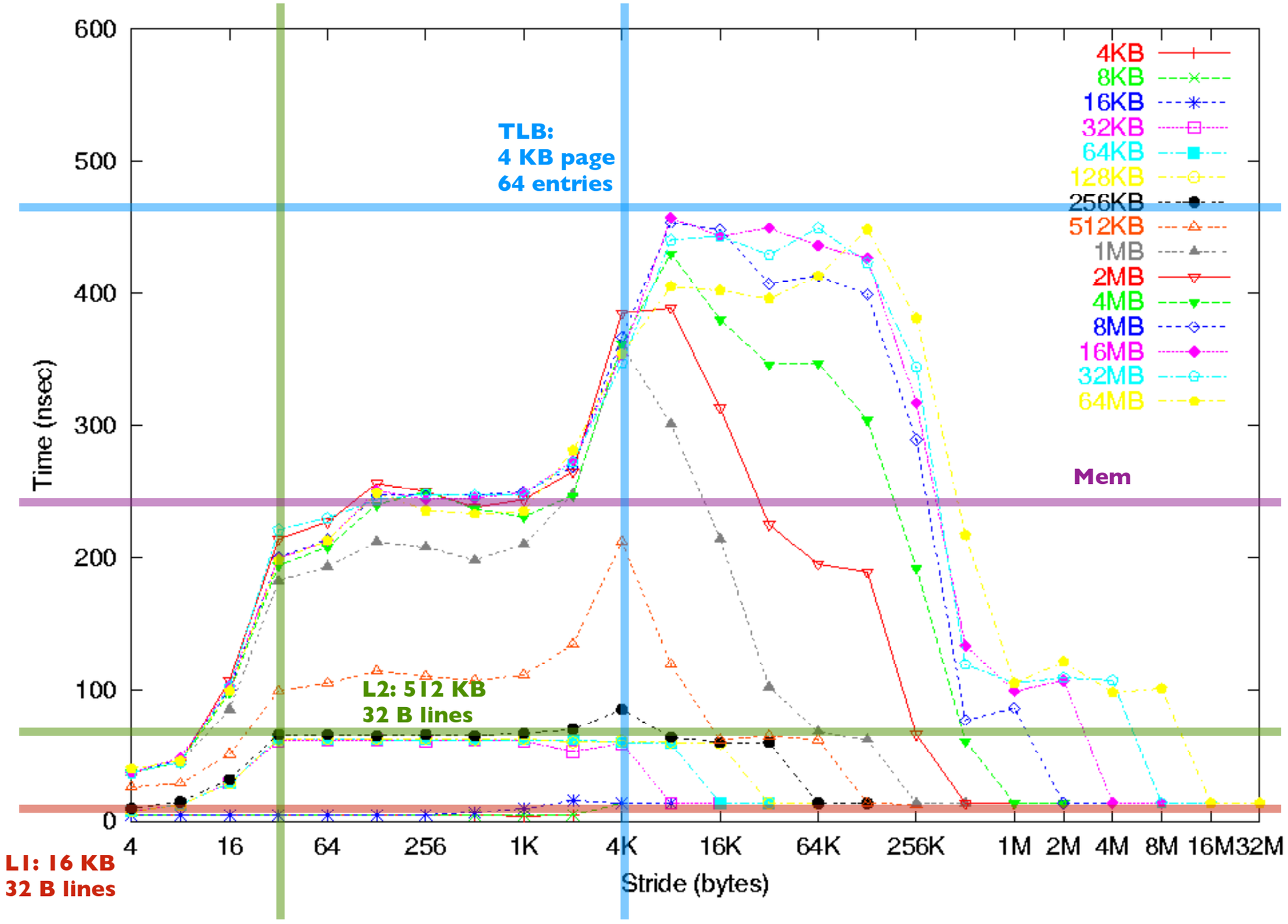
Average Memory Access Time (Saavedra-Barerra) — Sun Ultra IIi (333 MHz)



Average Memory Access Time (Saavedra-Barerra) — Sun Ultra IIi (333 MHz)

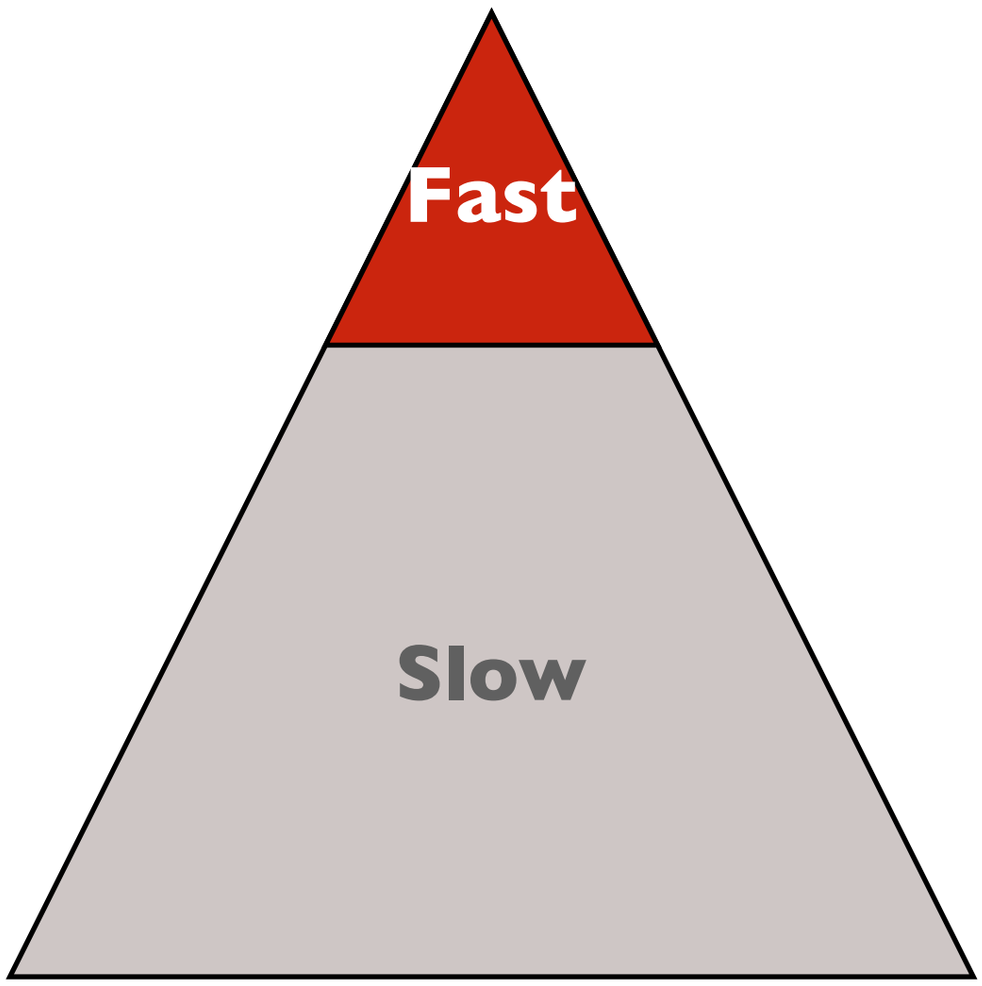


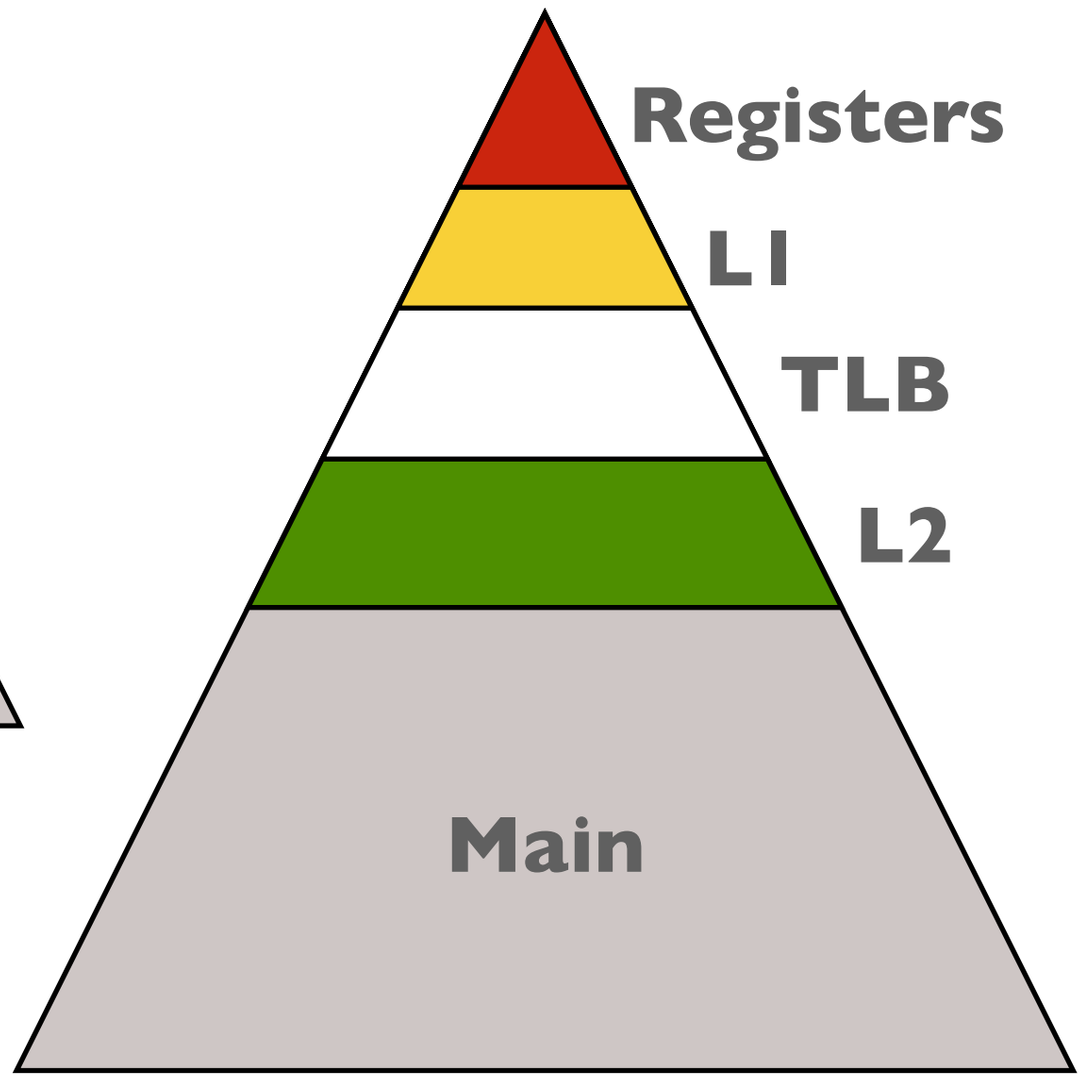
Average Memory Access Time (Saavedra-Barerra) — Pentium III (Katmai; 550 MHz)



"TLB" is part of the memory hierarchy

- Virtual address space – Why?
- Translation Look-aside Buffer helps manage virtual memory
  - Divide address space into pages (4–32 KB typical)
  - Page table maps virtual to physical addrs & whether page in mem or on disk
  - TLB caches 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

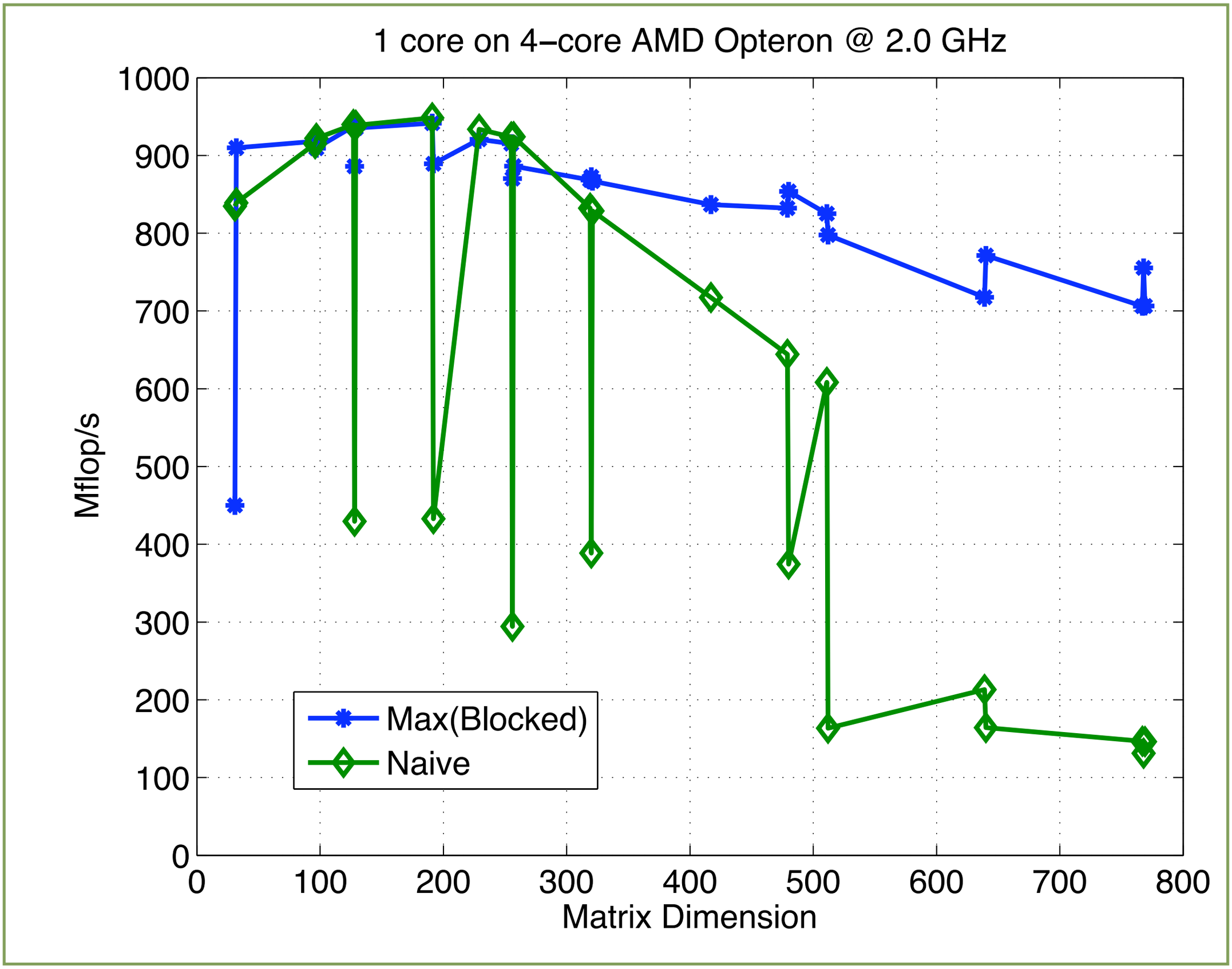

Fast
Slow


Registers
L I
TLB
L2
Main

- Bridging the gap between theory & practice
- Sources:
  - "Anatomy of high-performance matrix multiplication," by Goto & van de Geijn (2006/8)
  - "An experimental comparison of cache-oblivious and cache-conscious programs?" by Yotov, et al. (SPAA 2007)

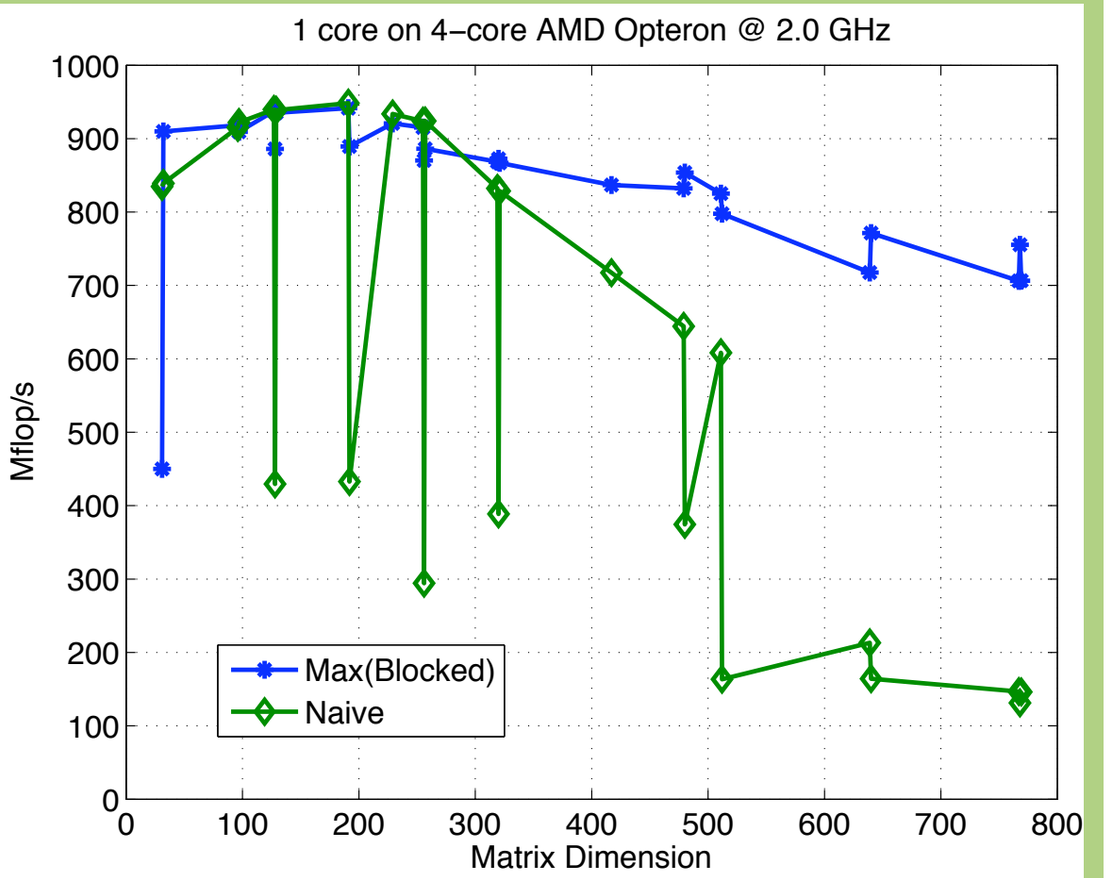
Sources

- "Anatomy of high-performance matrix multiplication," by Goto & van de Geijn (2006/8)
- "An experimental comparison of cache-oblivious and cache-conscious programs?" by Yotov, *et al.* (SPAA 2007)
- Course on tuning by Clint Whaley (UTSA)
- Course on tuning by Markus Püschel


1 core on 4–core AMD Opteron @ 2.0 GHz
Mflop/s
Matrix Dimension
Max(Blocked)
Naive
0
100
200
300
400
500
600
700
800
900
1000

Some questions

- Block size selection?
- Why the dips?
- For which level(s) of memory hierarchy do we optimize explicitly?



Block size constraints

- Assume:
  - Assume 1 FMA / cycle ⇒ ideal time = n^3 cycles
  - Square b x b blocks
  - Two-level memory (slow & fast)
- Recall capacity constraint: Lower bound

$$b \leq \sqrt{\frac{Z}{3}}$$

Upper-bound: Bandwidth constraint

$$
\begin{aligned}
f &\equiv \text{flops} = 2n^3 \\
m &\equiv \text{mops} \approx \frac{2n^3}{b} \\
\beta &\equiv \text{Peak bandwidth (words/cycle)} \\
\rho &\equiv \text{Peak flop/cycle} \\
\frac{m}{\beta} \leq \frac{f}{\rho} &\implies b \geq \frac{\rho}{\beta} \\
&\Downarrow \\
\frac{\rho}{\beta} \leq b &\leq \sqrt{\frac{Z}{3}}
\end{aligned}
$$

Block size constraint

$$\frac{\rho}{\beta} \leq \quad b \quad \leq \sqrt{\frac{Z}{3}}$$

$$\Downarrow$$

$$\begin{aligned} \rho &\sim 4 \ \frac{\text{flop}}{\text{cycle}} \\ \beta &\sim .4 \ \frac{\text{double-words}}{\text{cycle}} \quad (6.4 \text{ GB/s at } 2 \text{ GHz}) \\ Z &\sim 256 \text{ K-dwords (2 MB cache)} \end{aligned}$$

$$\Downarrow$$

$$10 \leq \quad b \quad \lesssim 295$$

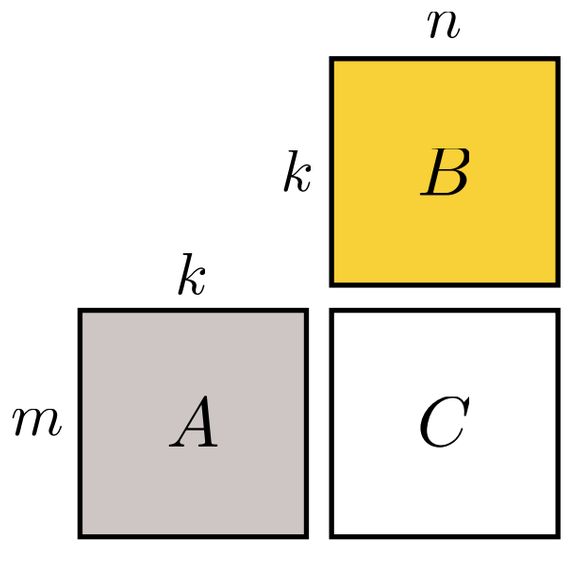
Multi-level blocking

- Idea: Block at multiple levels
  - Each cache, TLB, and CPU registers
  - Match multi-level memory hierarchy

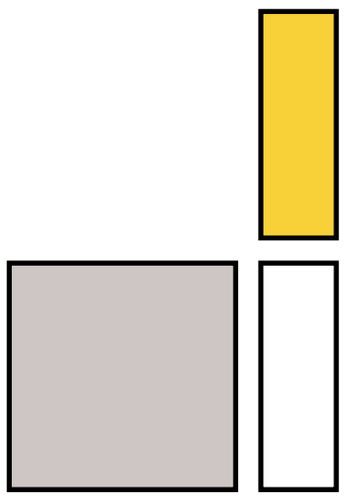
- Following discussion follows:
  "Anatomy of high-performance matrix multiplication," by Goto and van de Geijn (2006)

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

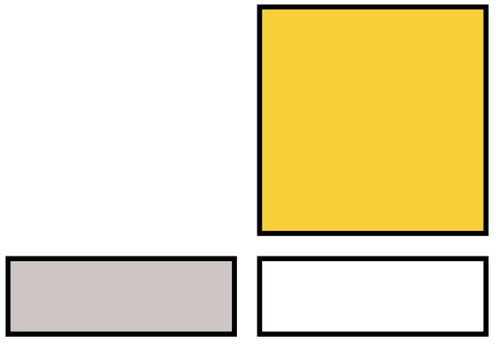
“Matrix-matrix”



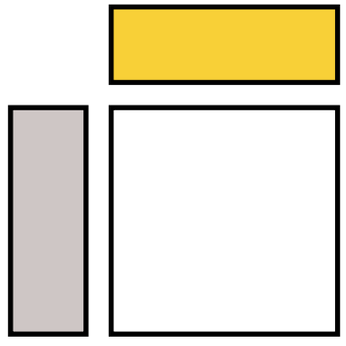
“Matrix-panel”

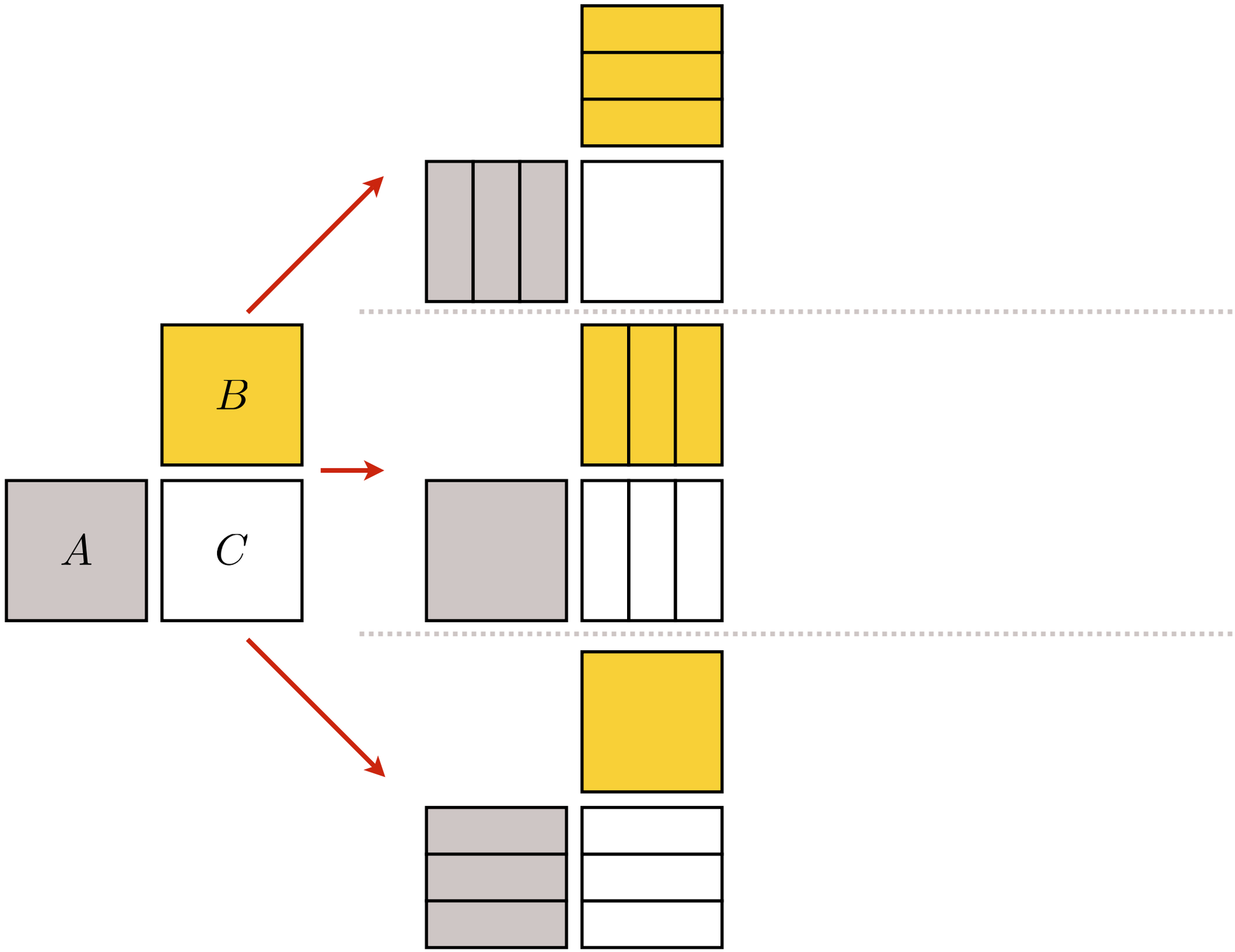


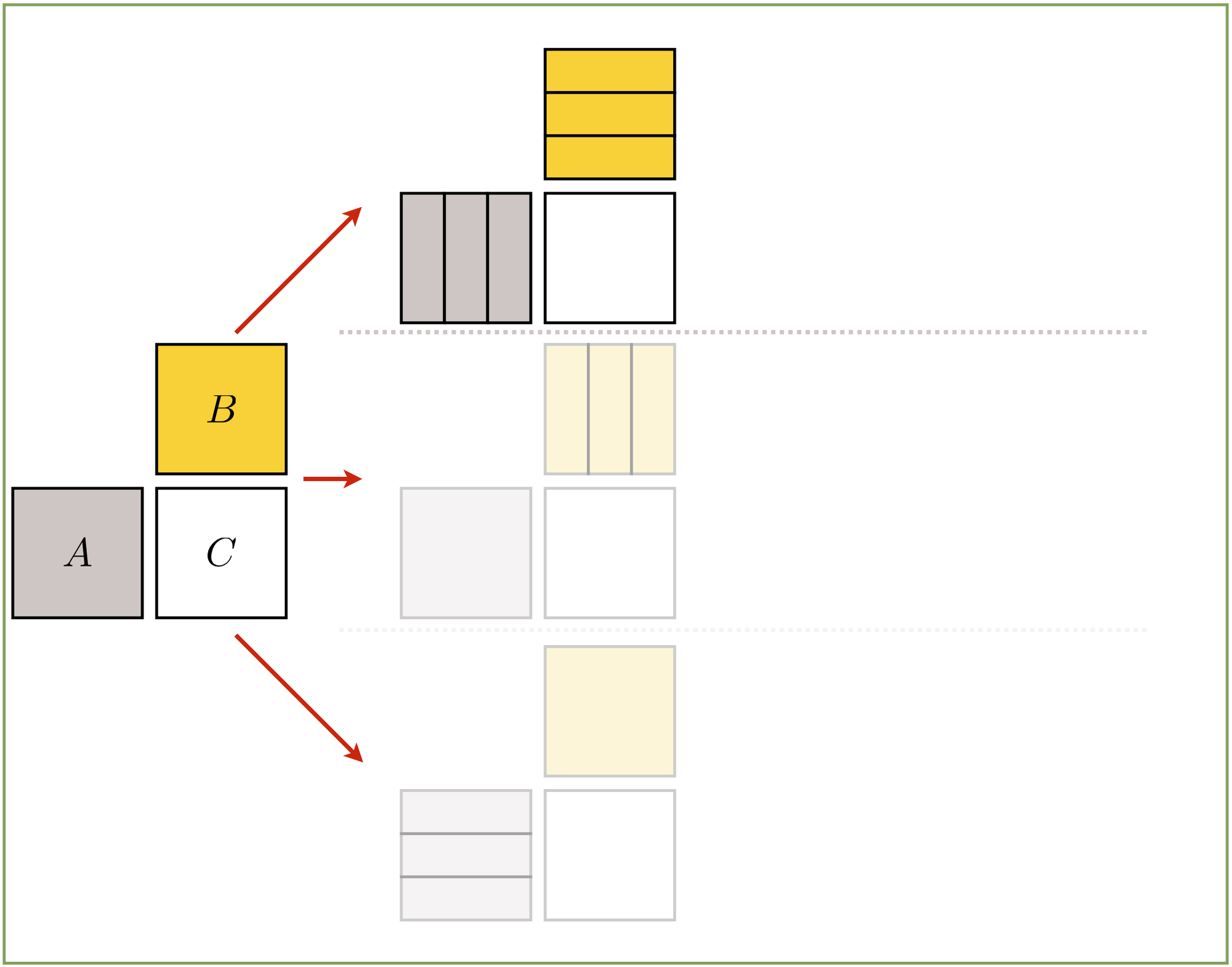
“Panel-matrix”

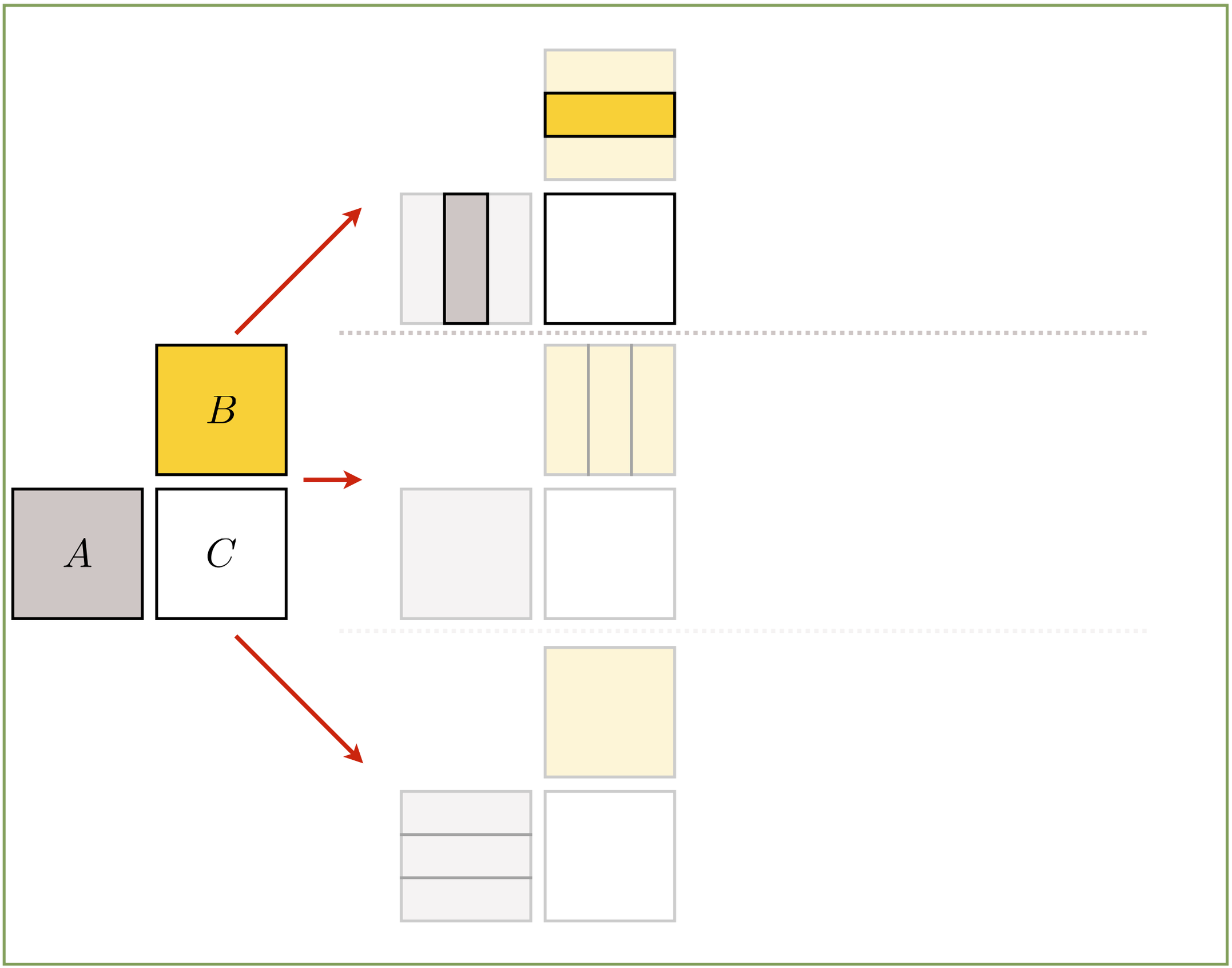


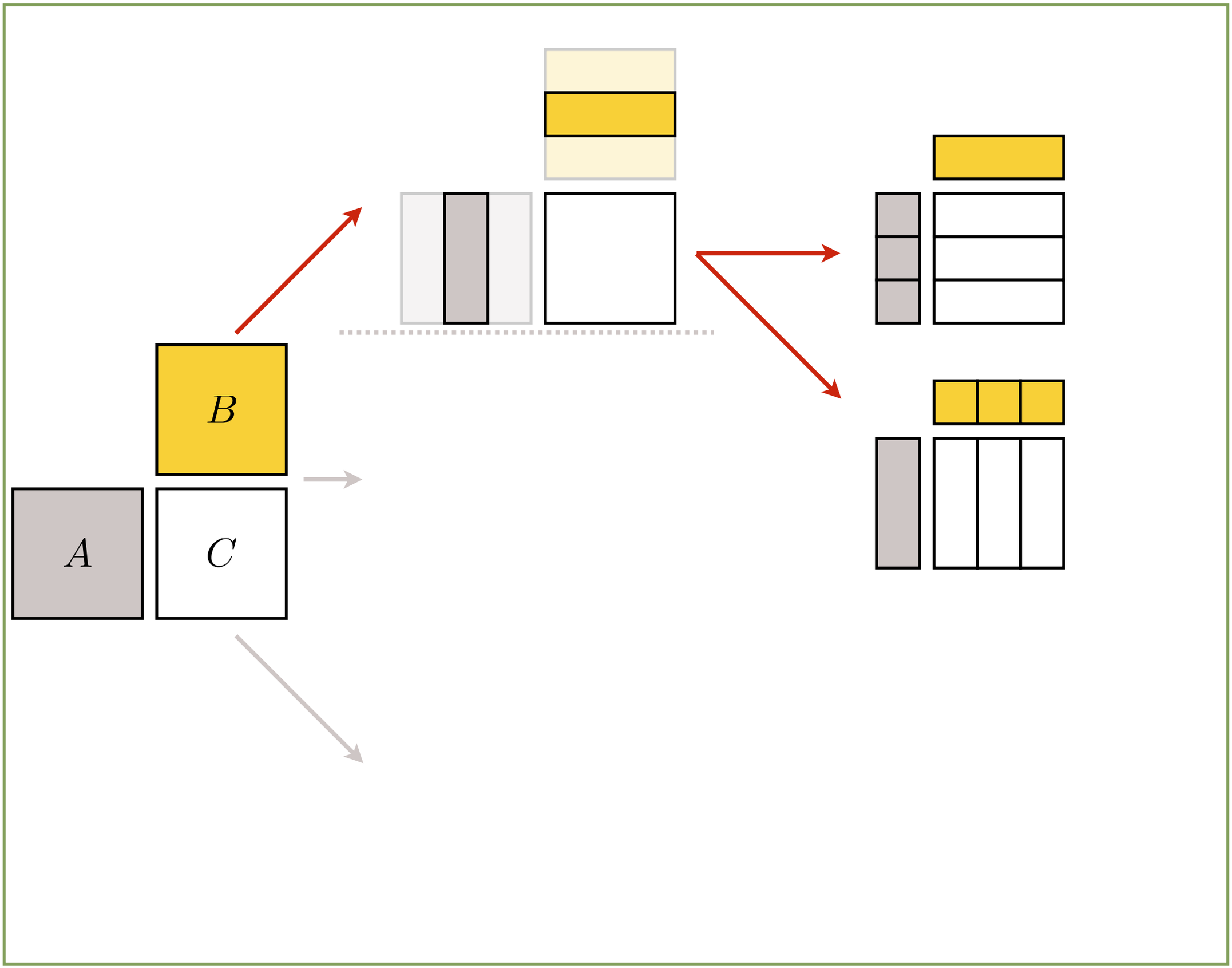
“Panel-Panel” or “Fat Outer Product”




B
A
C

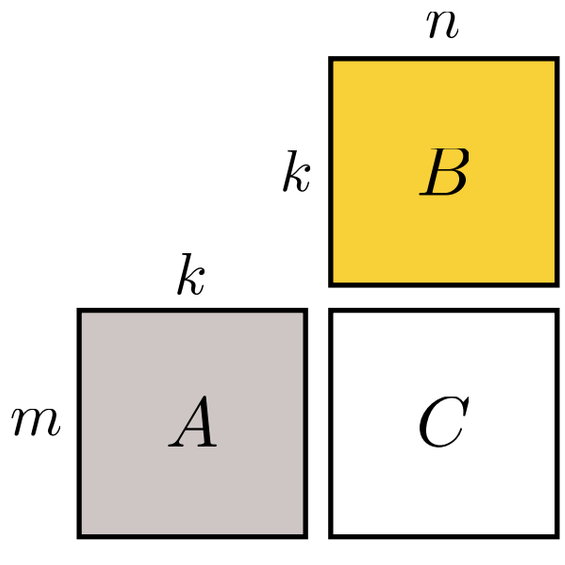

B
A
C


B
A
C

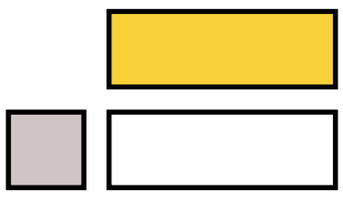

B
A
C

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

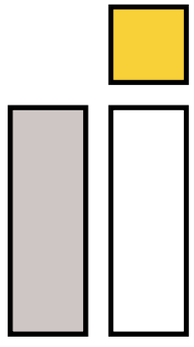
“Matrix-matrix”



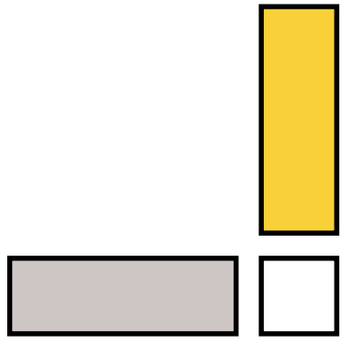
“Block-Panel”

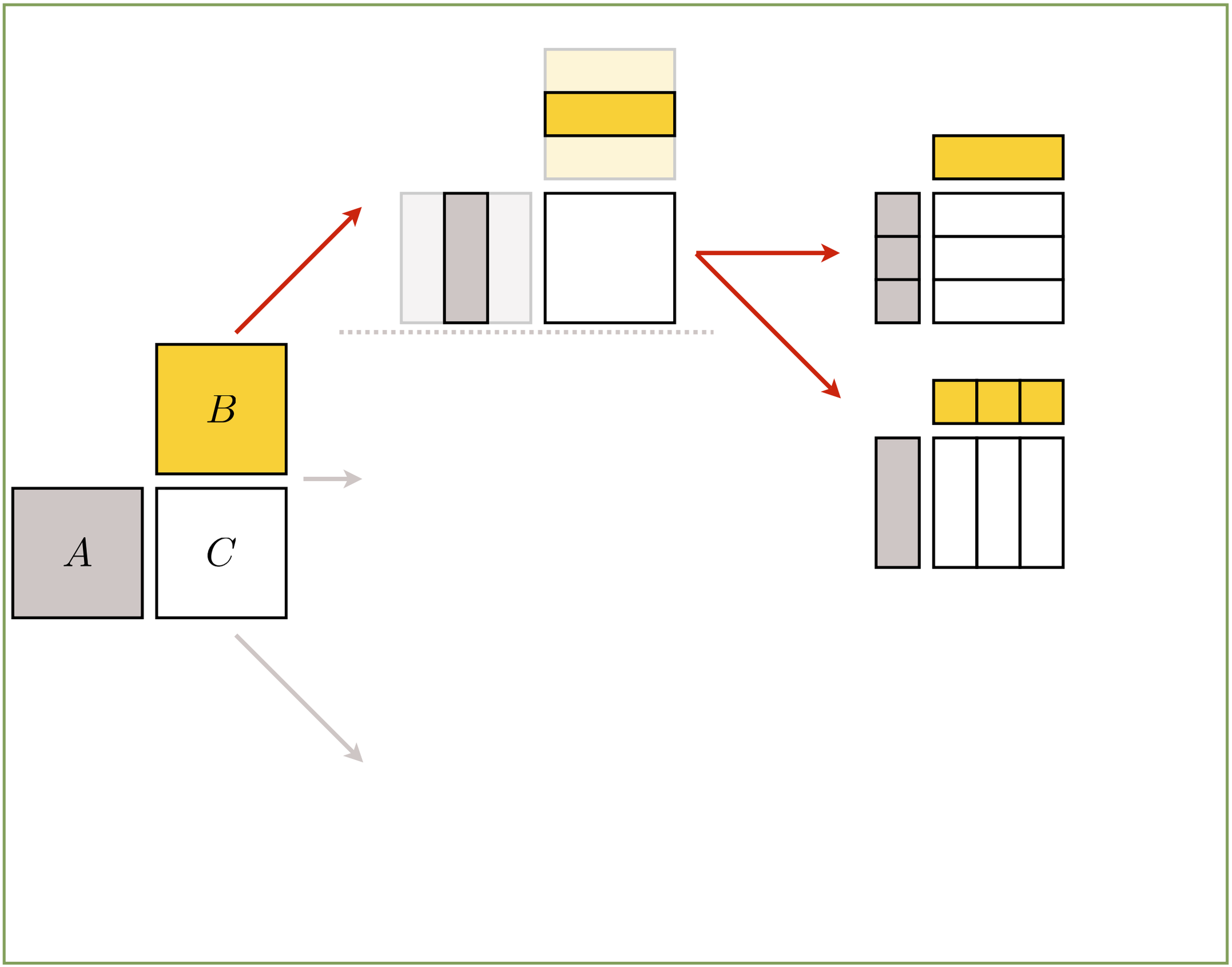


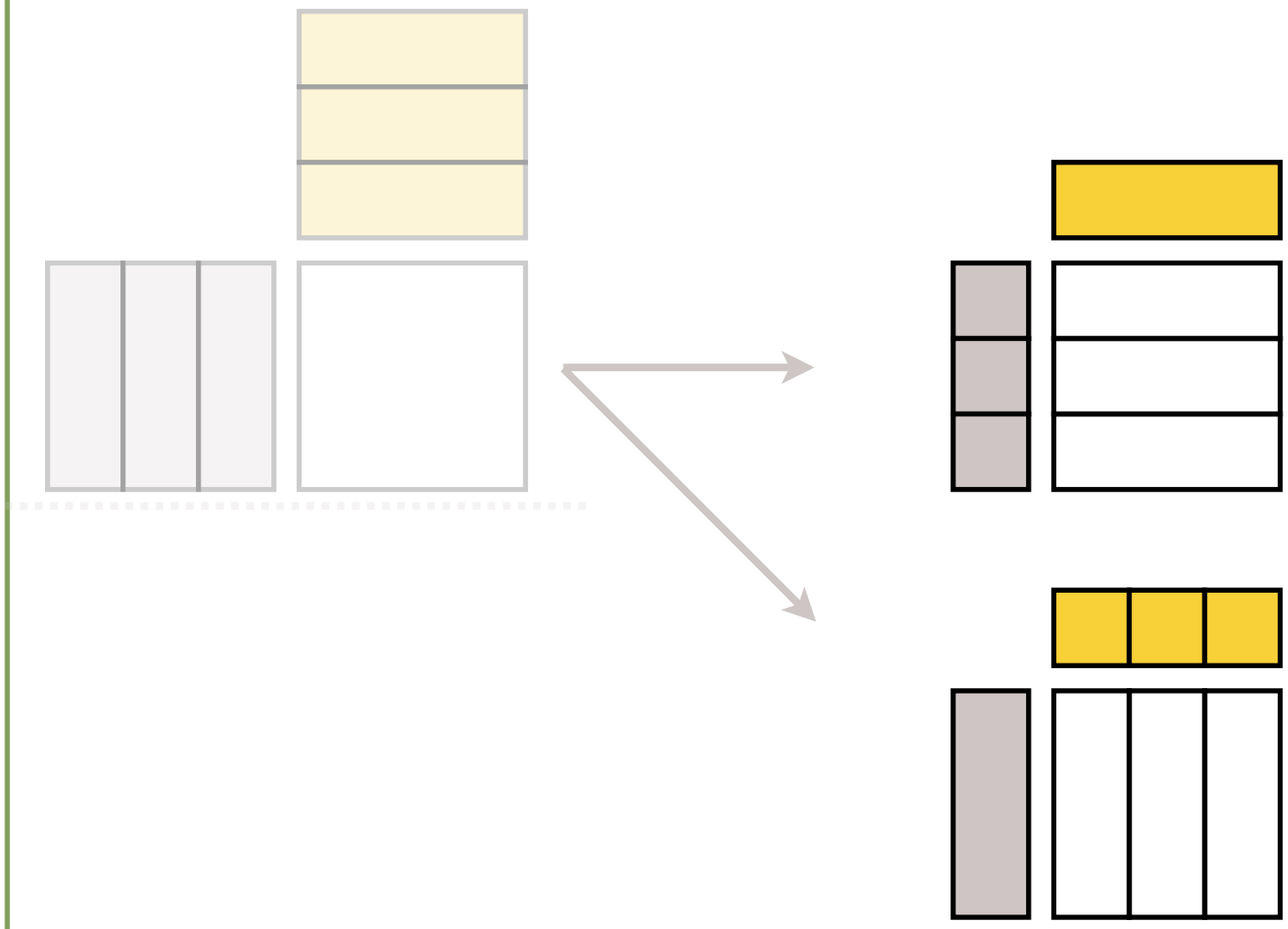
“Panel-block”

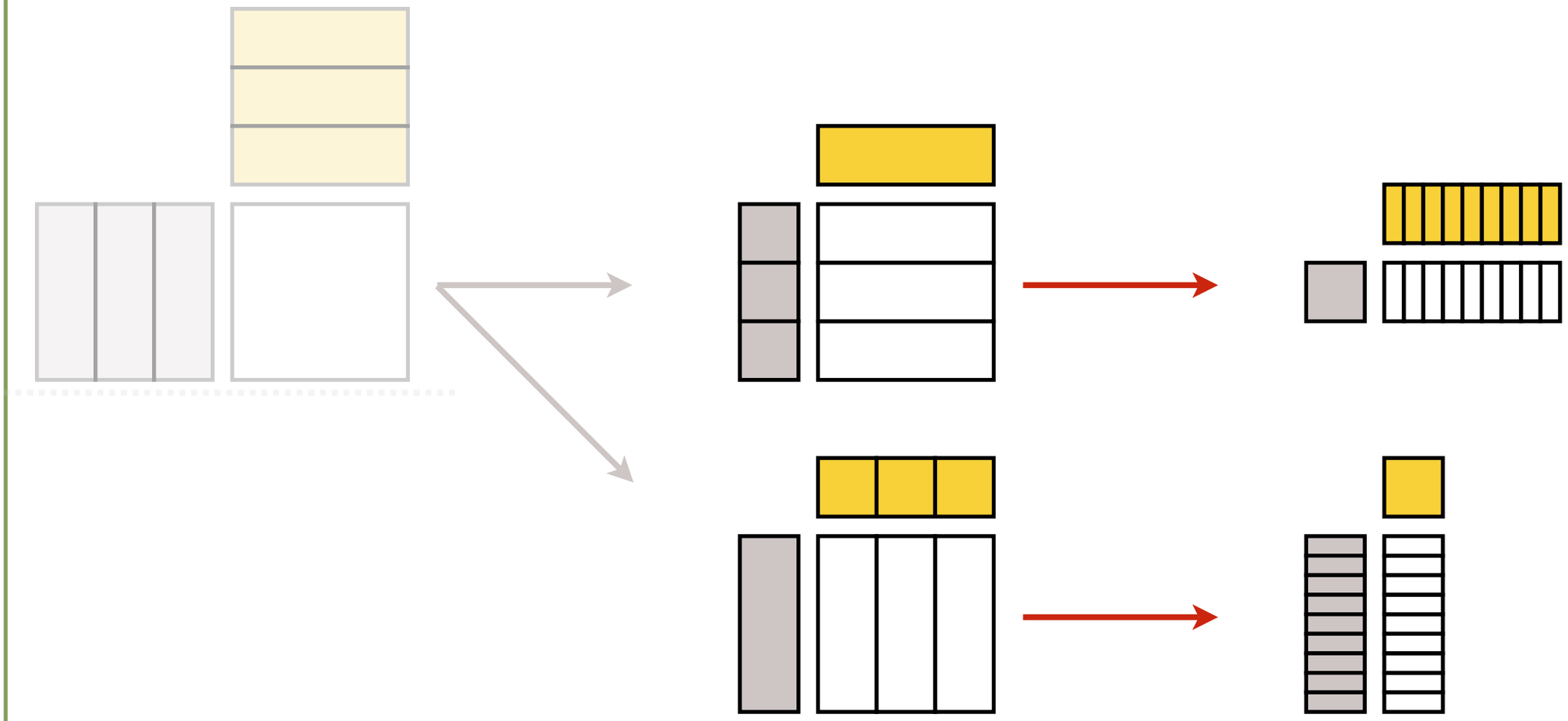


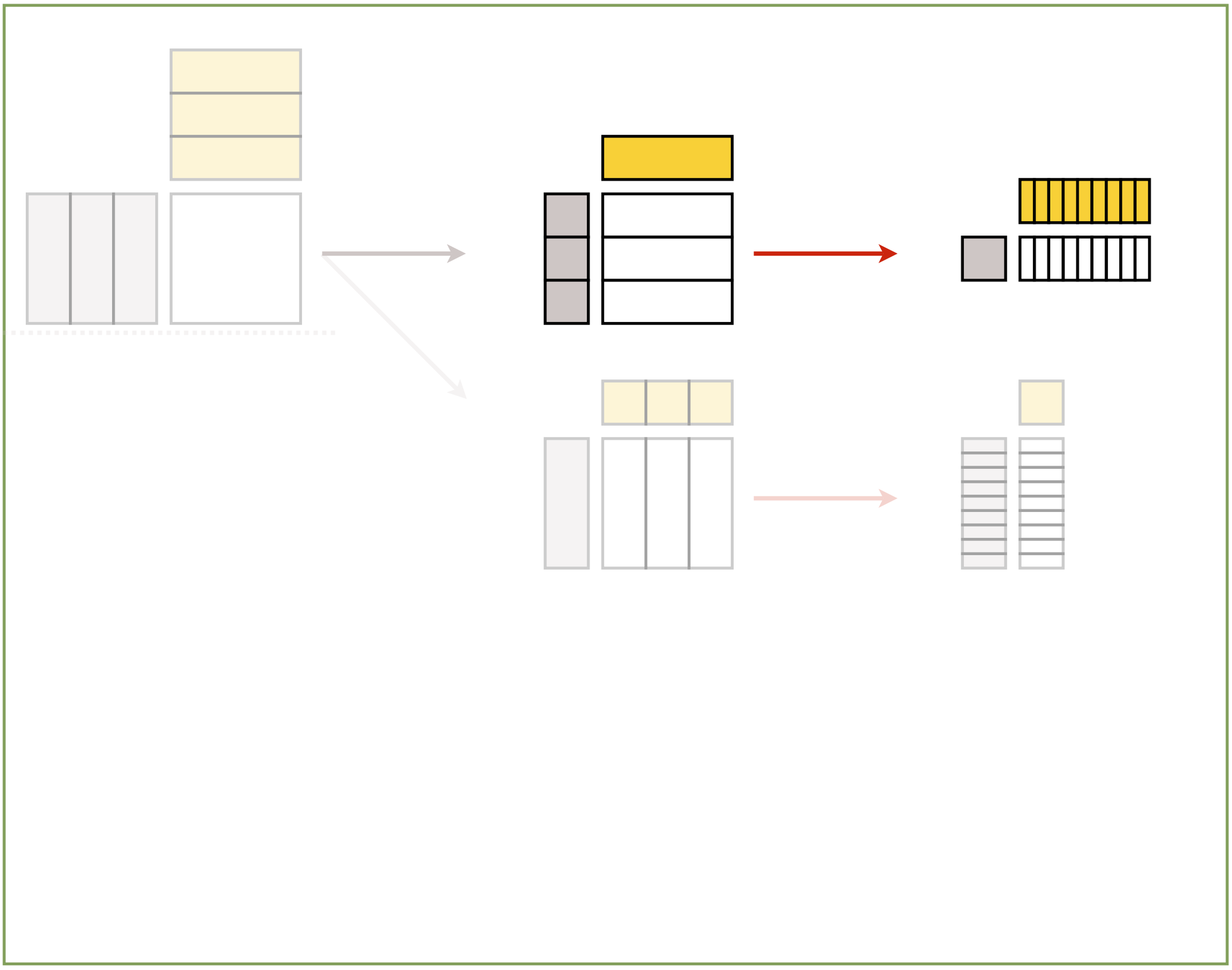
“Fat Dot Product”

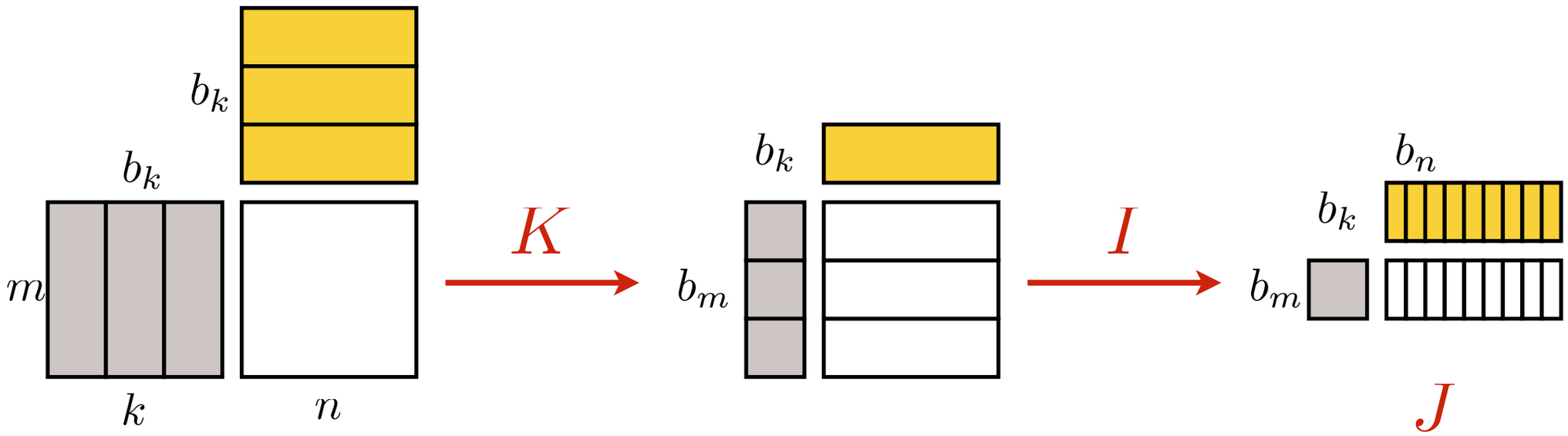



B
A
C









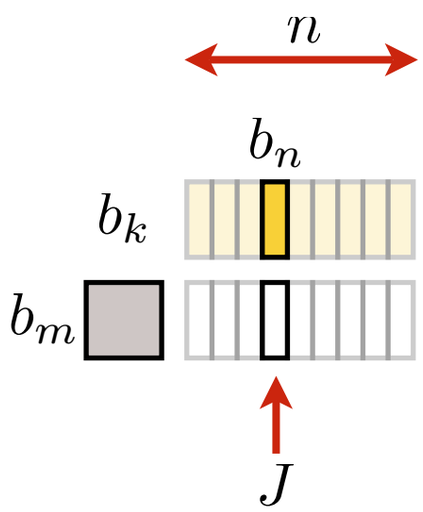
// Let I, J, K = blocks of indices

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

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

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

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



// "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

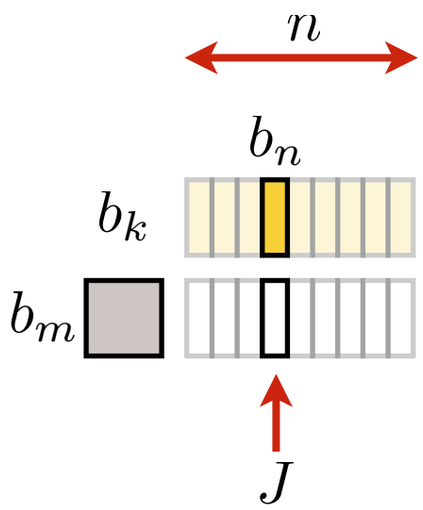
  $C_J \leftarrow C_J + A \times B_J$

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

Assumes:

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

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

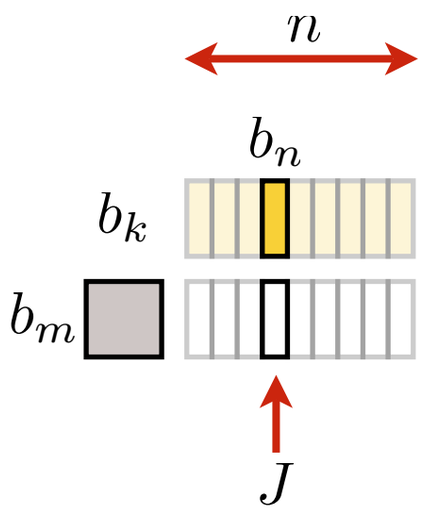


// "Block-panel" multiply
// Load $b_m \times b_k$ block of A into cache
for $J \leftarrow$ blocks 1 **to** $\frac{n}{b_n}$ **do**
  // Load $b_k \times b_n$ block of B into cache
  // Load $b_m \times b_n$ block of C into cache
  $C_J \leftarrow C_J + A \times B_J$
  // Store $b_m \times b_n$ block of C to memory

Assumes:
1. *A*, B_J, C_J fit in cache (e.g., size *Z*)
2. Above ⇒ Product runs at peak
3. *A* not evicted prematurely

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

$$\begin{aligned} f &= 2 b_m b_k n \\ m &= b_m b_k + (b_k + 2 b_m)\, n \\ &\Downarrow \\ q &= \frac{2}{\frac{1}{n} + \left(\frac{1}{b_m} + \frac{2}{b_k}\right)} \end{aligned}$$



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:

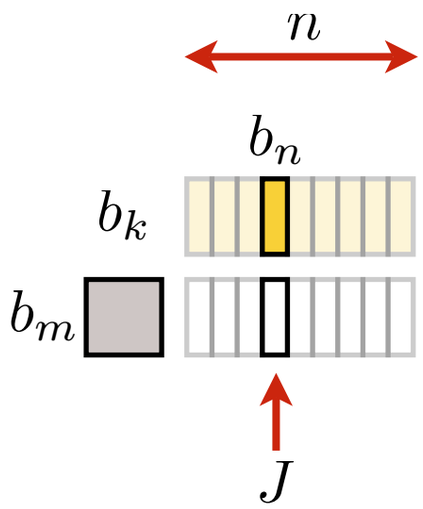
1. *A*, B_J, C_J fit in cache (e.g., size *Z*)
2. Above ⇒ Product runs at peak
3. *A* not evicted prematurely

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

$$\begin{aligned} \rho_1 &\equiv \text{Peak L1 flop/s} \\ \beta_2 &\equiv \text{Peak L2 to CPU bw} \end{aligned}$$

$$\begin{aligned} \frac{2 b_m b_k b_n}{\rho_1} &\geq \frac{b_m b_k}{\beta_2} \\ &\Downarrow \\ b_n &\geq \frac{\rho_1}{2\beta_2} \end{aligned}$$

Typically, need b_n >= 2.

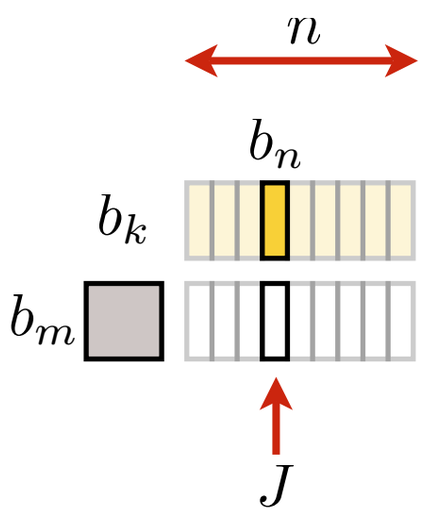


Assumes:

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

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

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



Assumes:

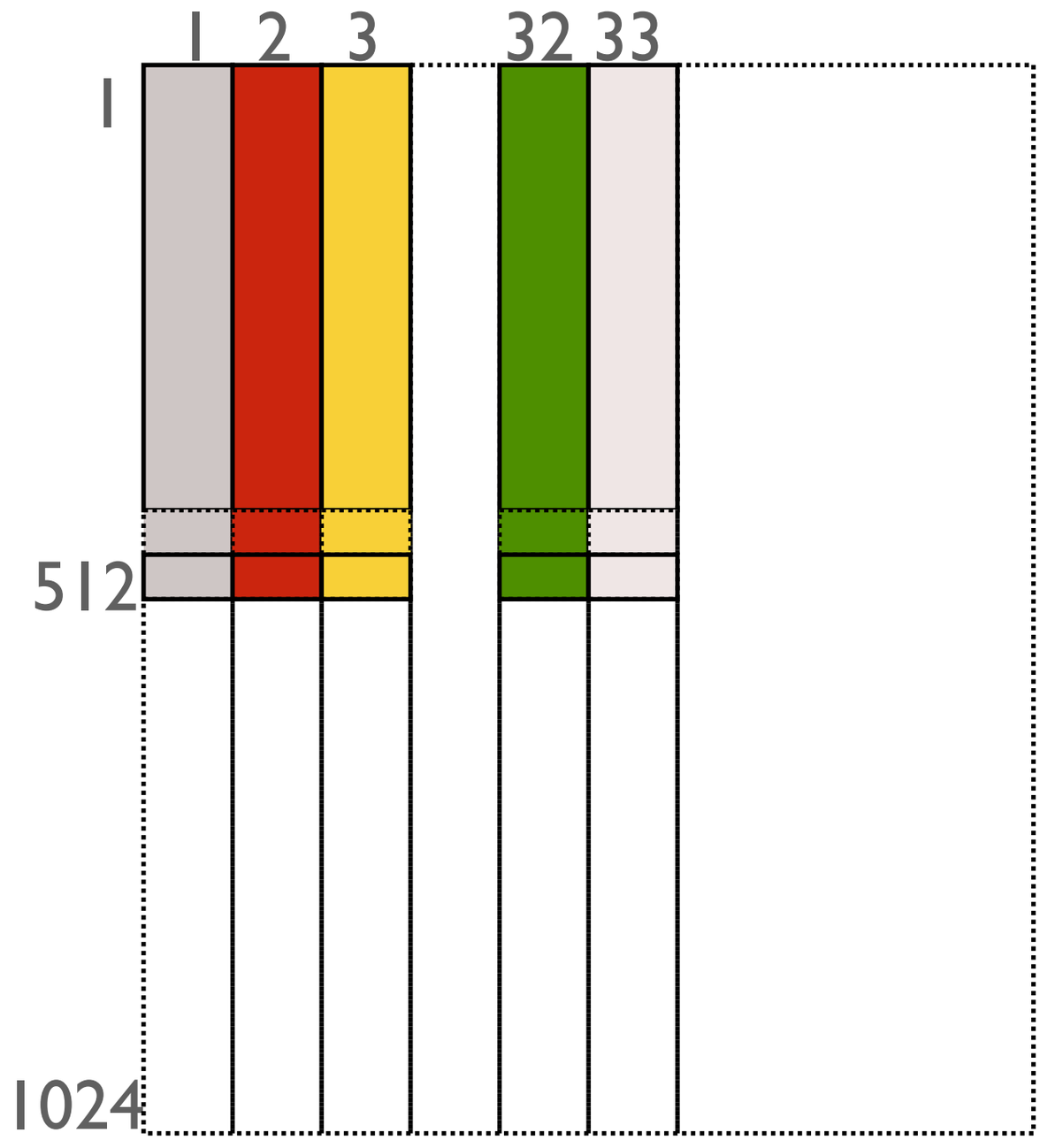
1. *A*, B_J, C_J fit in cache (e.g., size *Z*)
2. Above ⇒ Product runs at peak
3. *A* not evicted prematurely

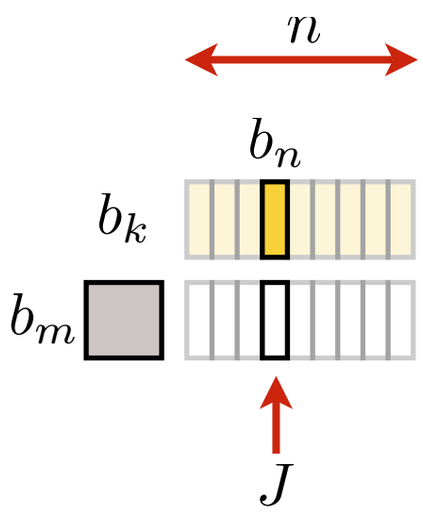
$$\begin{aligned} b_m b_k + (b_k + b_m) b_n &\leq Z \\ b_n &\geq \frac{\rho_1}{2\beta_2} \end{aligned}$$

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 *Z*)
2. Above ⇒ Product runs at peak

3. *A* not evicted early
4. Operands "fit in" TLB

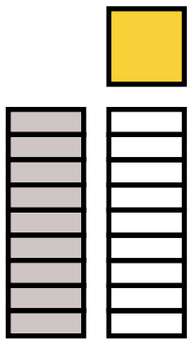
$$\begin{aligned} b_m b_k + (b_k + b_m) b_n &\leq Z \\ b_n &\geq \frac{\rho_1}{2\beta_2} \end{aligned}$$

What about the TLB?

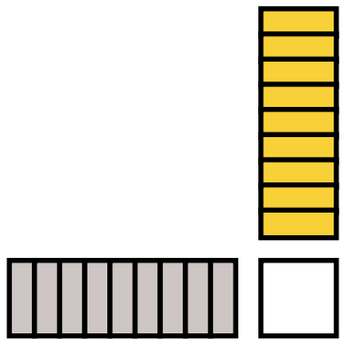
Block of *A* straddles pages, so re-pack on-the-fly
⇒ **"Copy optimization"**

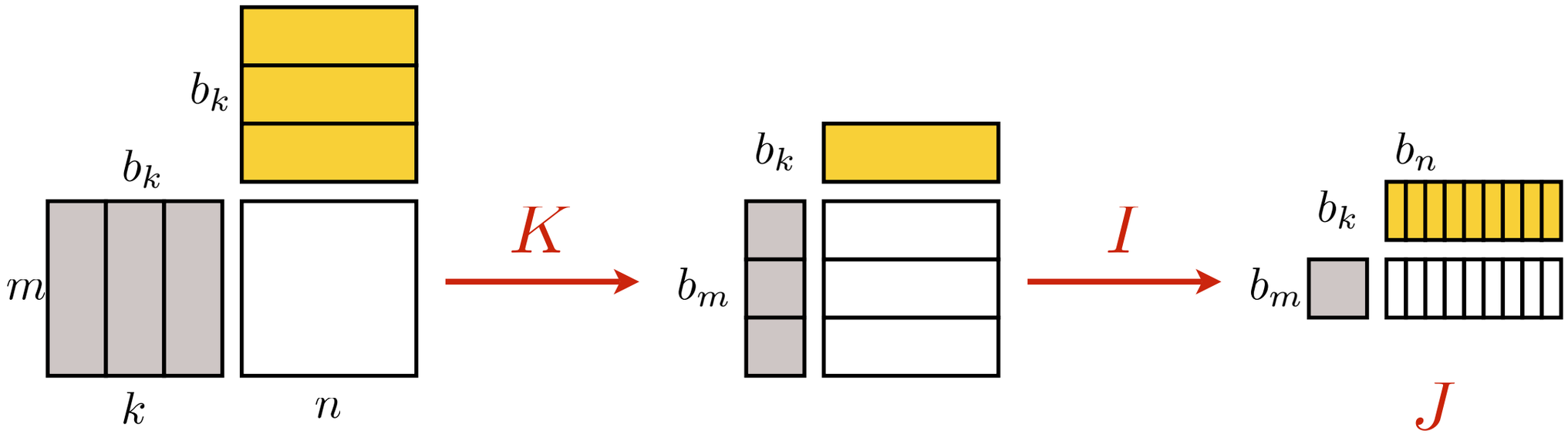
Copy *B* panel as well

Panel-Block



Fat-Dot





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

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

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

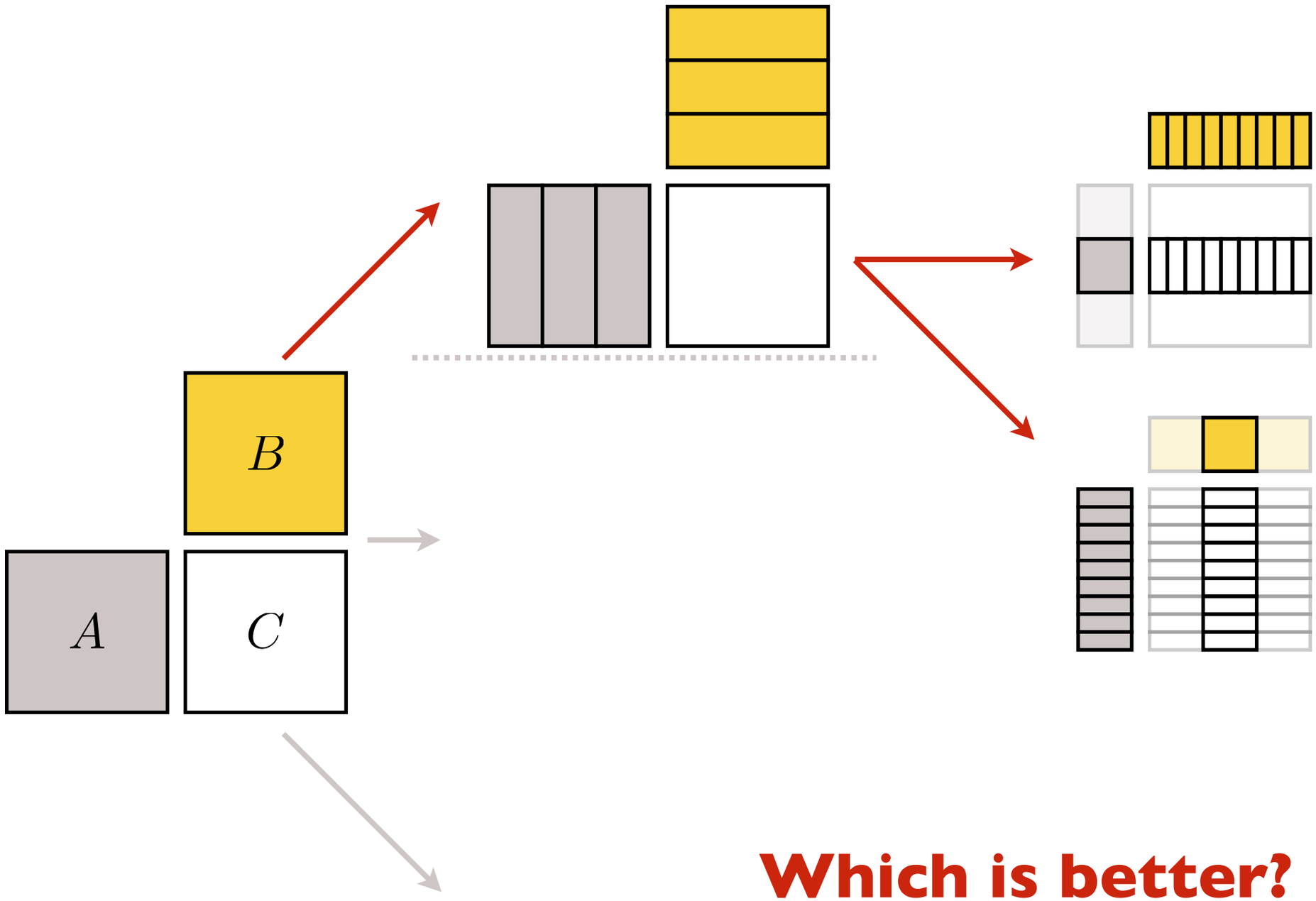
for $I \leftarrow$ blocks 1 **to** $\dfrac{m}{b_m}$ **do**

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

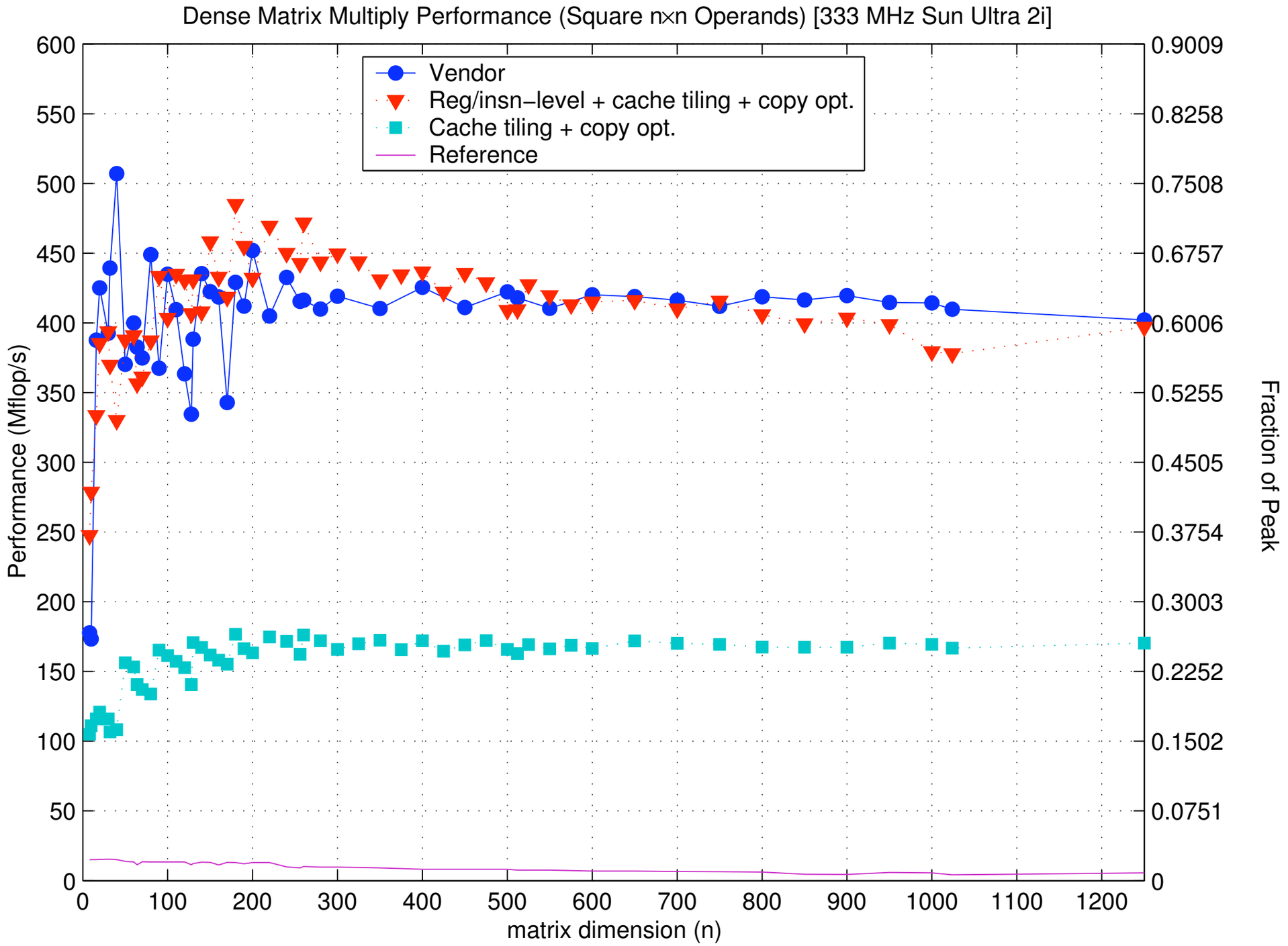
for $J \leftarrow$ blocks 1 **to** $\dfrac{n}{b_n}$ **do**

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

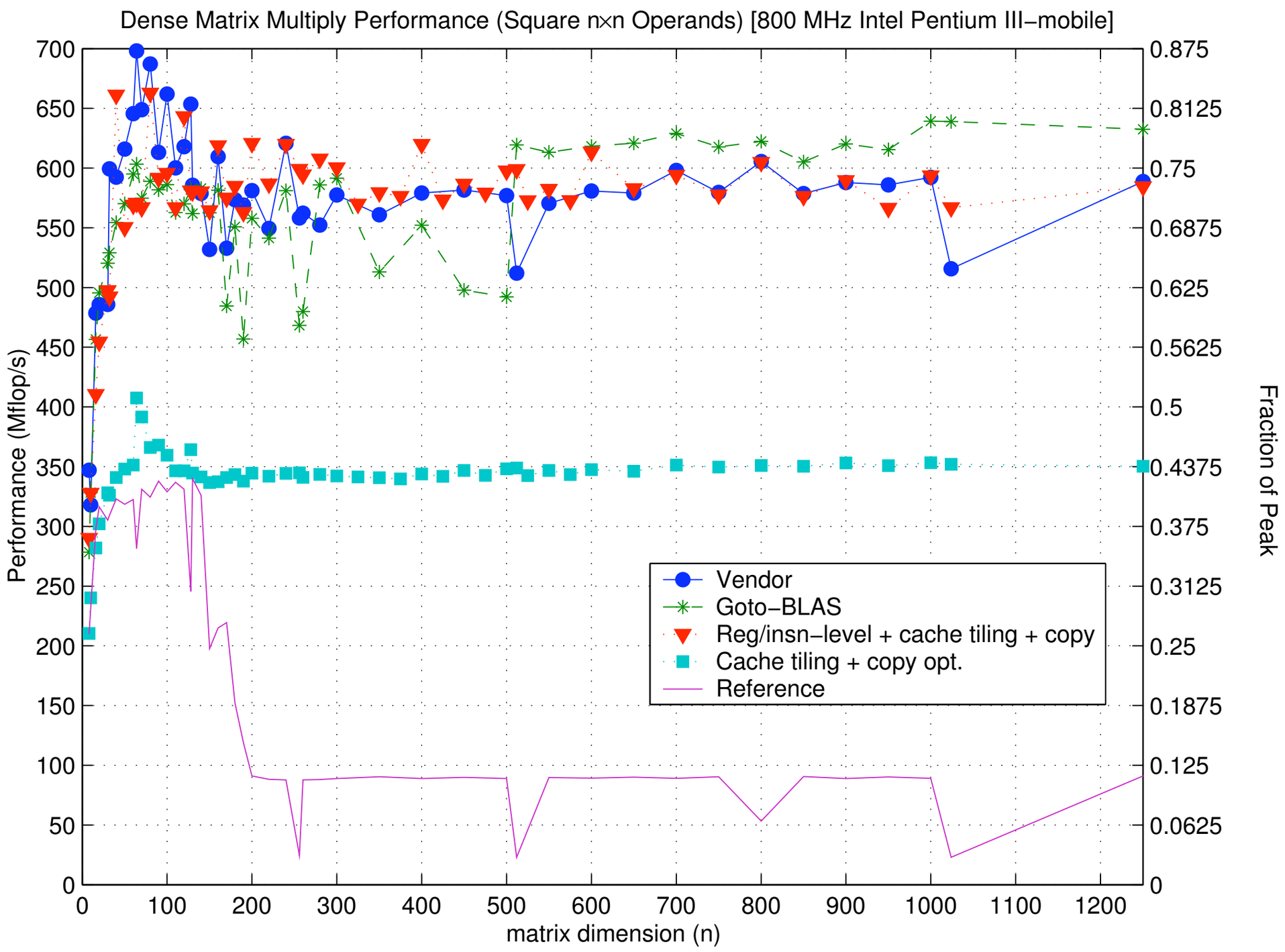
$C_{IJ} \leftarrow C_{IJ} + \tilde{C}$ // Unpack $\tilde{C}$



Which is better?



Source:Vuduc, Demmel, Bilmes (IJHPCA 2004)

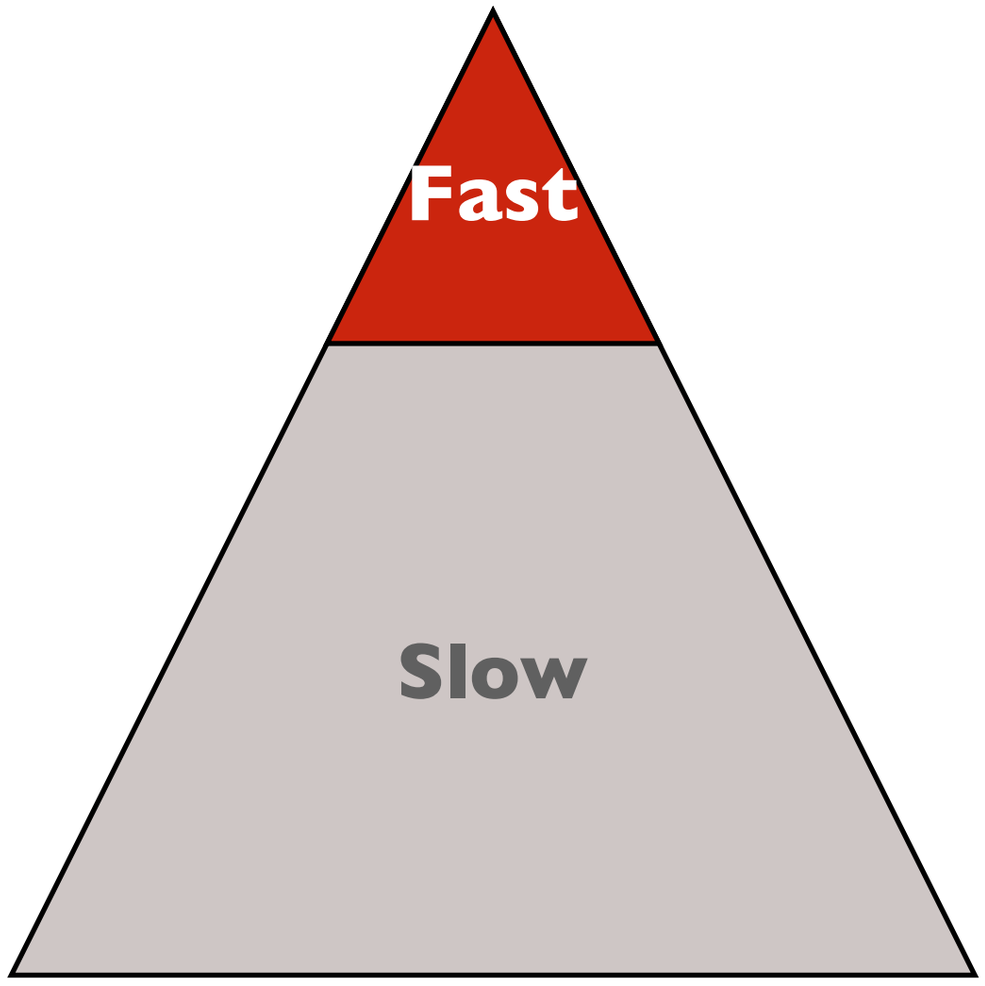


Source:Vuduc, Demmel, Bilmes (IJHPCA 2004)

- Cache-oblivious matrix multiply (from Yotov, *et al.* SPAA '07)

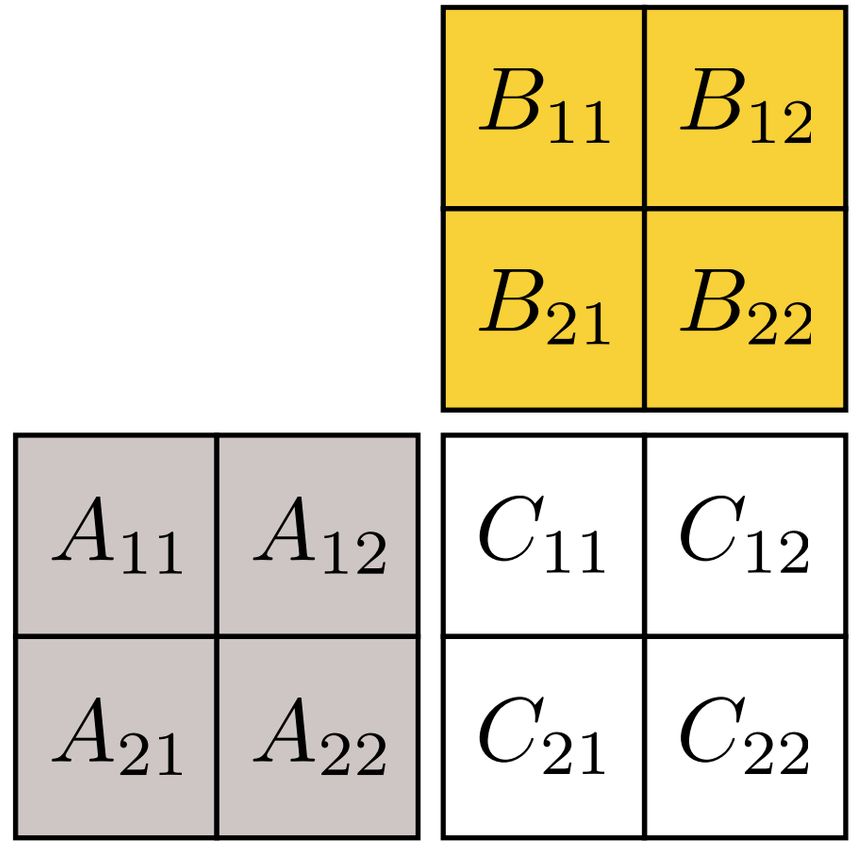
Memory model for analyzing cache-oblivious algorithms

- Two-level memory hierarchy
- Z = capacity of cache ("fast")
- L = cache line size
- Fully associative
- Optimal replacement
  - Evicts most distant use
  - Sleator & Tarjan (CACM 1985): LRU, FIFO w/in constant of optimal w/ cache larger by constant factor
- "Tall-cache:" $Z \geq O(L^2)$
  - Limits: See Brodal & Fagerberg (STOC 2003)
  - When might this not hold?



A recursive algorithm for matrix-multiply

- Divide all dimensions in half
- Bilardi, *et al.*: Use Gray code ordering

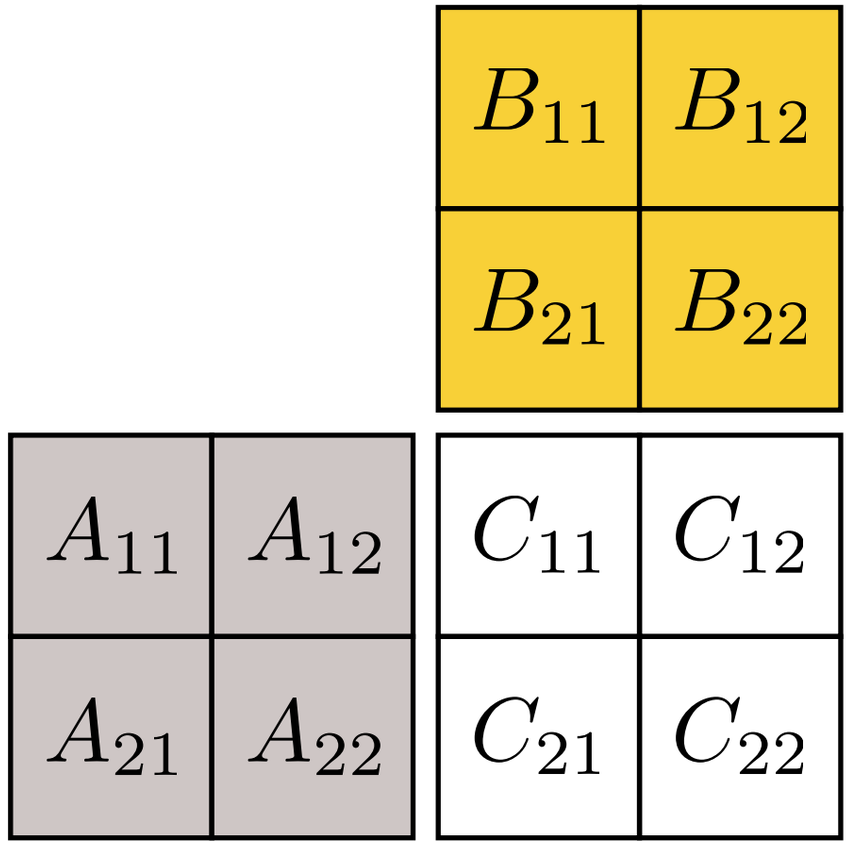


$$\text{Cost (flops)} = T(n) = \begin{cases} 8 \cdot T(\frac{n}{2}) & n > 1 \\ O(1) & n = 1 \end{cases}$$

$$= O(n^3)$$

A recursive algorithm for matrix-multiply

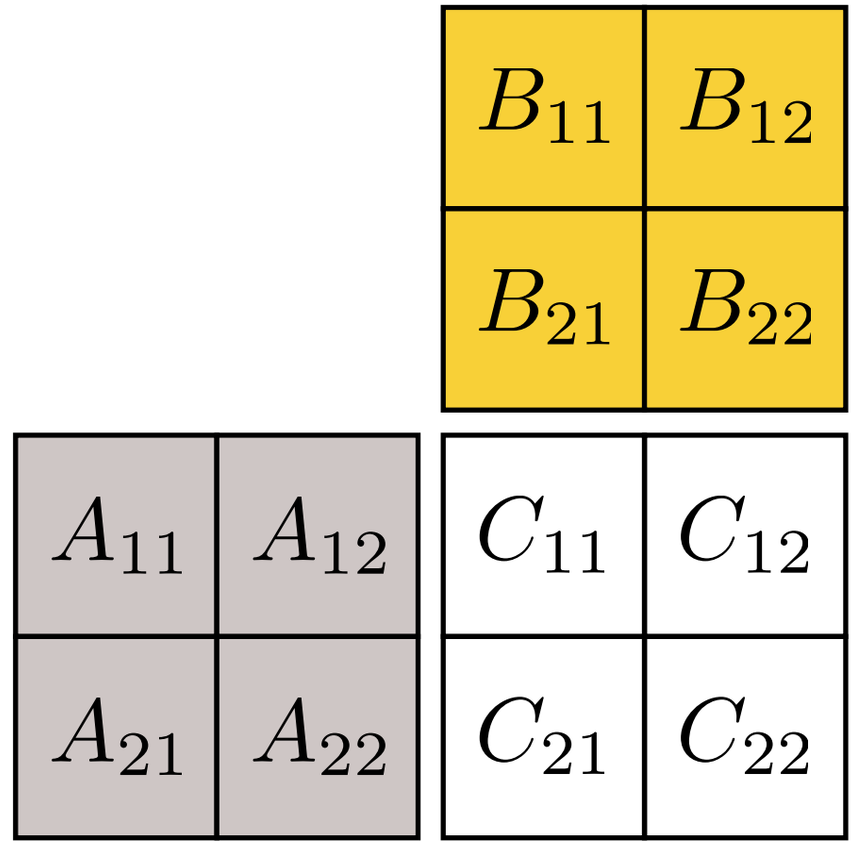
- Divide all dimensions in half
- Bilardi, *et al.*: Use Gray-code ordering



I/O Complexity?

A recursive algorithm for matrix-multiply

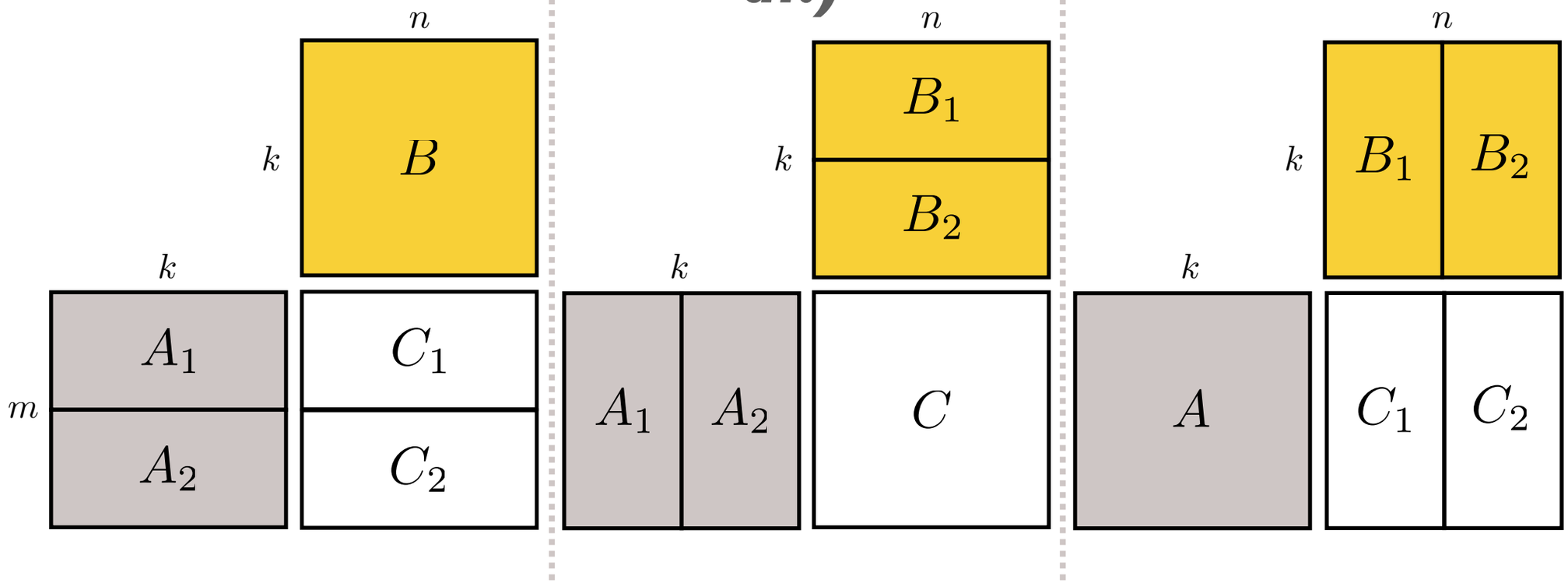
- Divide all dimensions in half
- Bilardi, *et al.*: Use Gray-code ordering



No. of misses, with tall-cache assumption:

$$Q(n) = \left\{ \begin{array}{ll} 8 \cdot Q(\frac{n}{2}) & \text{if } n > \sqrt{\frac{Z}{3}} \\ \frac{3n^2}{L} & \text{otherwise} \end{array} \right\} \leq \Theta\left(\frac{n^3}{L\sqrt{Z}}\right)$$

Alternative: Divide longest dimension (Frigo, *et al.*)

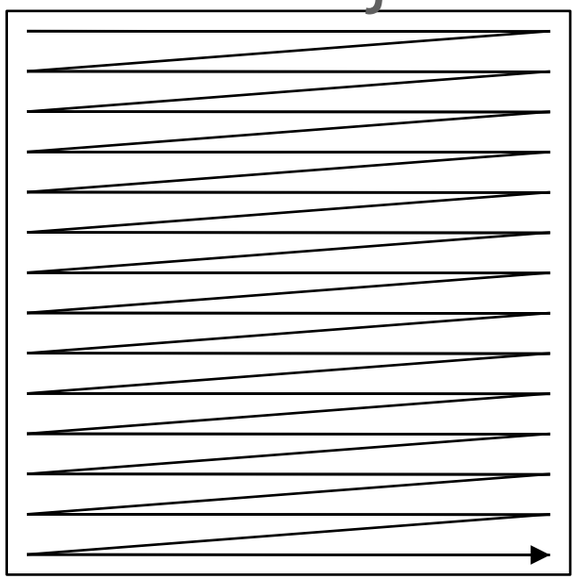


$$\text{Cache misses } Q(m,k,n) \quad \leq \quad \begin{cases} \Theta\left(\frac{mk+kn+mn}{L}\right) & \text{if } mk+kn+mn \leq \alpha Z \\ 2Q\left(\frac{m}{2},k,n\right) & \text{if } m \geq k,n \\ 2Q(m,\frac{k}{2},n) & \text{if } k > m, k \geq n \\ 2Q(m,k,\frac{n}{2}) & \text{otherwise} \end{cases}$$

$$= \quad \Theta\left(\frac{mkn}{L\sqrt{Z}}\right)$$

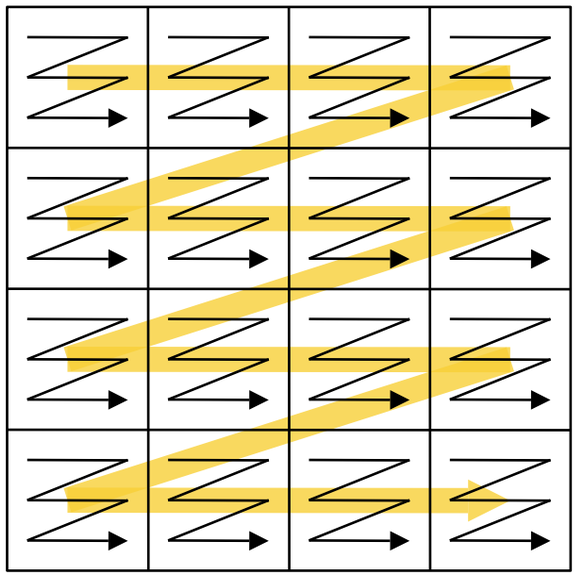
Relax tall-cache assumption using suitable layout

Row-major



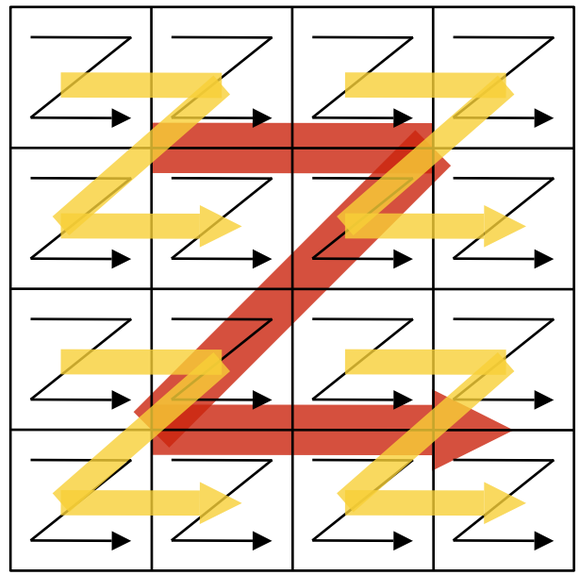
Need tall cache

Row-block-row



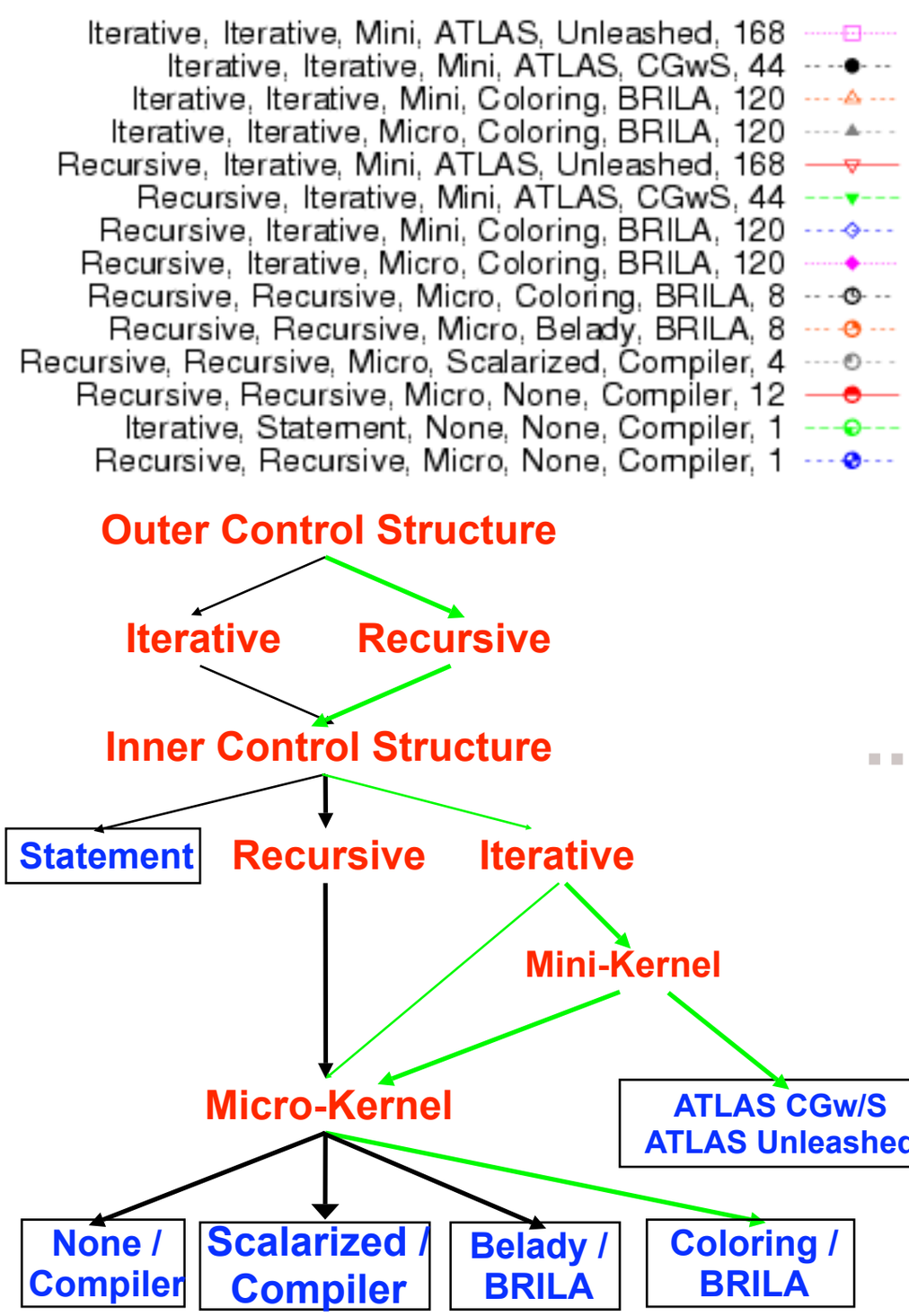
$M \geq \Omega(L)$

Morton Z

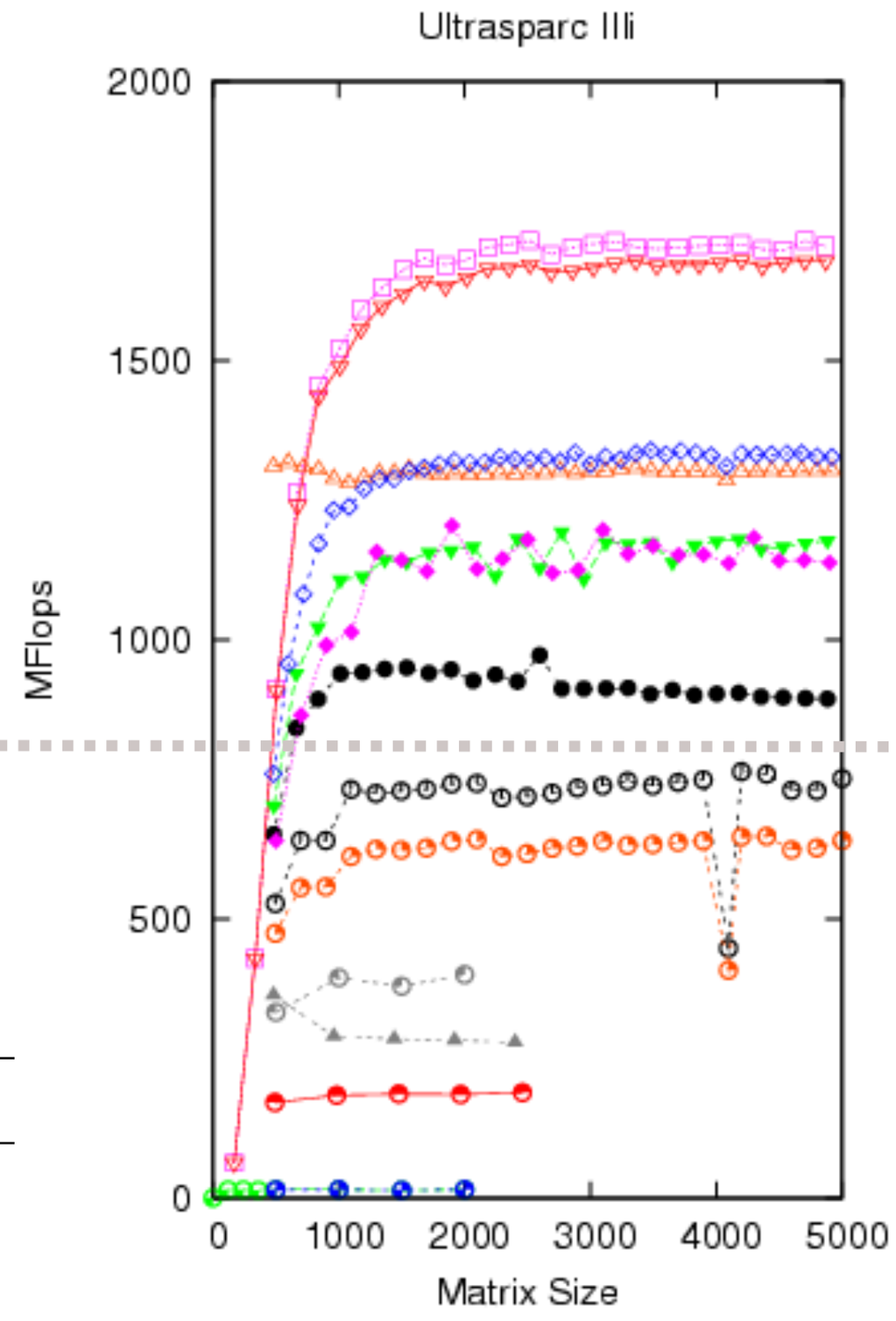


No assumption

Source: Yotov, et al. *(SPAA 2007) and Frigo,* et al. *(FOCS '99)*


Outer Control Structure
Iterative
Recursive
Inner Control Structure
Statement
Recursive
Iterative
Mini-Kernel
Micro-Kernel
ATLAS CGw/S
ATLAS Unleashed
None / Compiler
Scalarized / Compiler
Belady / BRILA
Coloring / BRILA

Iterative, Iterative, Mini, ATLAS, Unleashed, 168
Iterative, Iterative, Mini, ATLAS, CGwS, 44
Iterative, Iterative, Mini, Coloring, BRILA, 120
Iterative, Iterative, Micro, Coloring, BRILA, 120
Recursive, Iterative, Mini, ATLAS, Unleashed, 168
Recursive, Iterative, Mini, ATLAS, CGwS, 44
Recursive, Iterative, Mini, Coloring, BRILA, 120
Recursive, Iterative, Micro, Coloring, BRILA, 120
Recursive, Recursive, Micro, Coloring, BRILA, 8
Recursive, Recursive, Micro, Belady, BRILA, 8
Recursive, Recursive, Micro, Scalarized, Compiler, 4
Recursive, Recursive, Micro, None, Compiler, 12
Iterative, Statement, None, None, Compiler, 1
Recursive, Recursive, Micro, None, Compiler, 1


Ultrasparc IIIi
MFlops
2000
1500
1000
500
0
0
1000
2000
3000
4000
5000
Matrix Size

Summary: Cache-oblivious engineering considerations

- Need to cut-off recursion
- Careful scheduling/tuning required at "leaves"
- Yotov, *et al.*, report that full-recursion + tuned micro-kernel ≤ 2/3 best
- Open issues
  - Recursively-scheduled kernels worse than iteratively-schedule kernels — why?
  - Prefetching needed, but how best to apply in recursive case?