Serial optimization

Compiler options

	Intel	GNU
Compiler	ifort,icpc,icpc	gfortran,gcc,g++
Options	-Ofast,-ipa	-fast
Libraries	-ipa	

Compiler options

- Test multiple options
  - -fast is not always the fastest
  - The higher the optimization, the greater the chance of error (generally only an issue with pointers)

Some of the ways compilers optimize code

- Loop unrolling
- Loop interchange
- Vectorizing operations
- Loop unswitching
- Loop nest optimizations
- Loop invariant code motion
- Inlining
- Machine-specific optimization

Why optimization techniques are important

- Optimized code often runs at least 5 times faster than unoptimized code (a factor of 50 is not unheard of)
- The compiler can't always recognize when it can apply an optimization technique, so it becomes your responsibility
- If the compiler is too aggressive in its optimizations, it can create bugs

Anatomy of a loop

```
for(I=0; I < n; I++){
Out[I]=in[I];
}
```

```
I=0
do{
If(I >= n) break;
Out[I]=in[I];
I=I+1;
}
```

Loop unrolling

```
for(I=0; I < n; I++){
  out[I]=in[I];
}
```

ORIGINAL

```
for(I=0;I< n; I+=4){
  out[I]=in[I];
  out[I+1]=in[I+1];
  out[I+2]=in[I+2];
  out[I+3]=in[I+3];
}
```

Unrolled

Less jumps and conditional checks

Longer code, takes up more registers

Loop unrolling

```
I=0
If I < n break;
Out[I]=in[I]
I=I+1
If I < n break;
Out[I]=in[I]
I=I+1
If I < n break;
Out[I]=in[I]
I=I+1
If I < n break;
Out[I]=in[I]
I=I+1
```

```
I=0
If I < n break;
Out[I]=in[I]
Out[I+1]=in[I+1]
Out[I+2]=in[I+2]
Out[I+3]=in[I+3]
I=I+4
```

Loop interchange

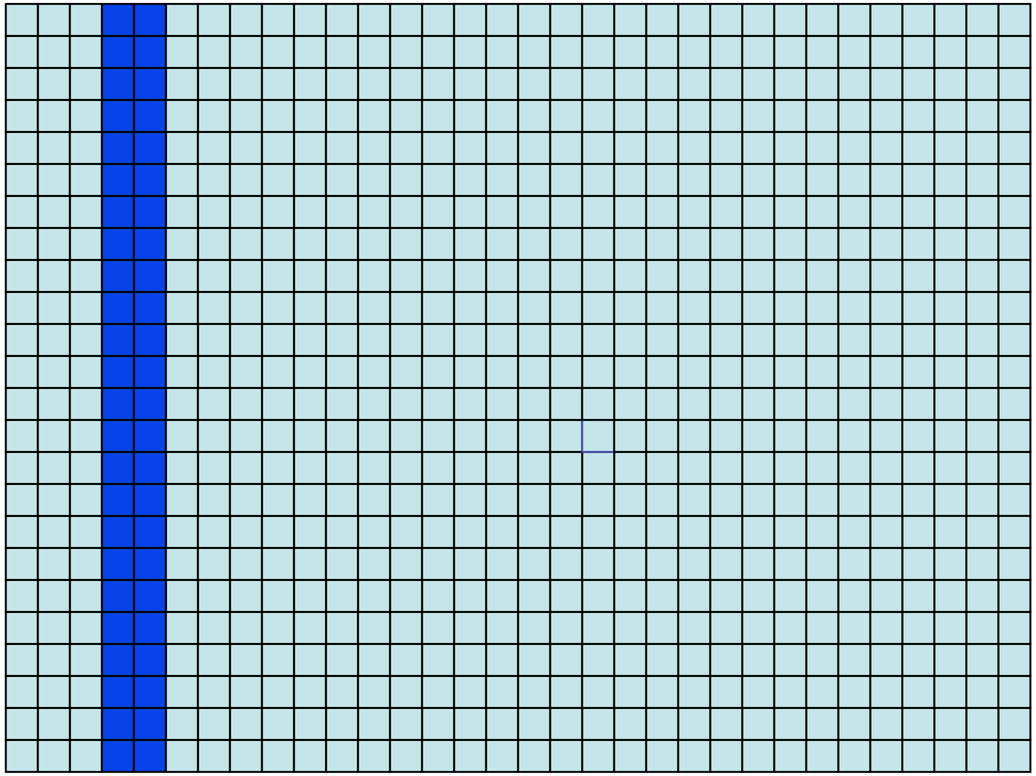
```
for(j=0; j<m; j++){
  for(I=0; I < n; I++){
  out[I][j]=in[I][j];
}}
```

ORIGINAL

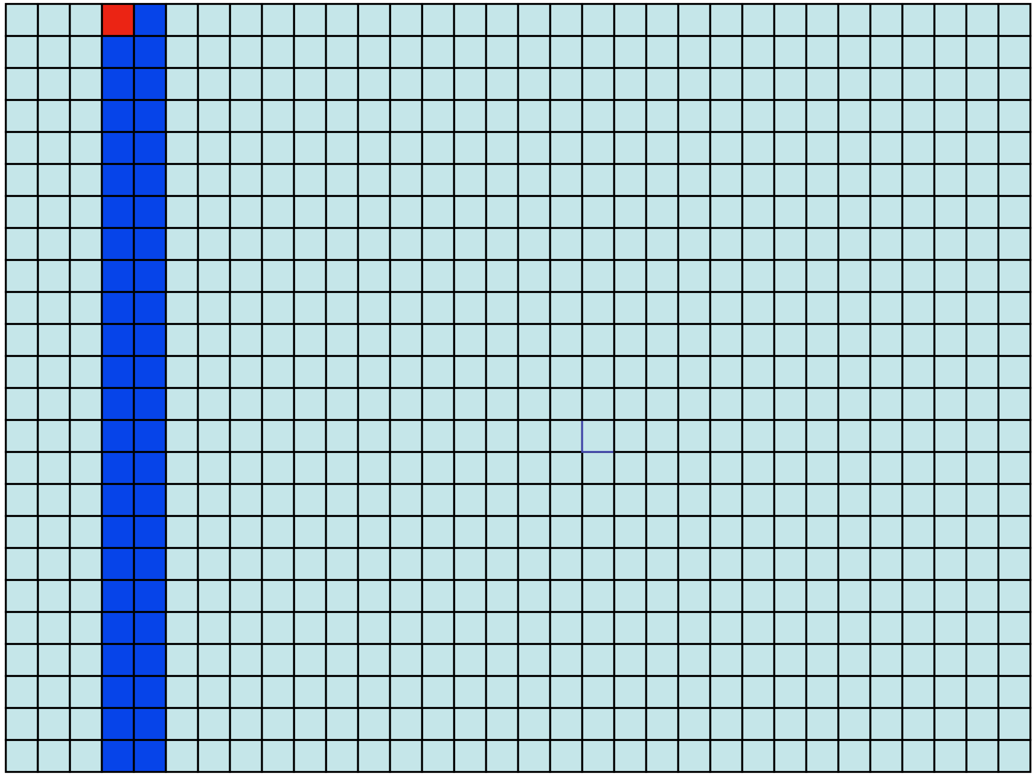
```
for(I=0; I<n; I++){
  for(j=0; j < m; j++){
  out[I][j]=in[I][j];
}}
```

Interchange loop order to avoid cache misses

Loop interchange

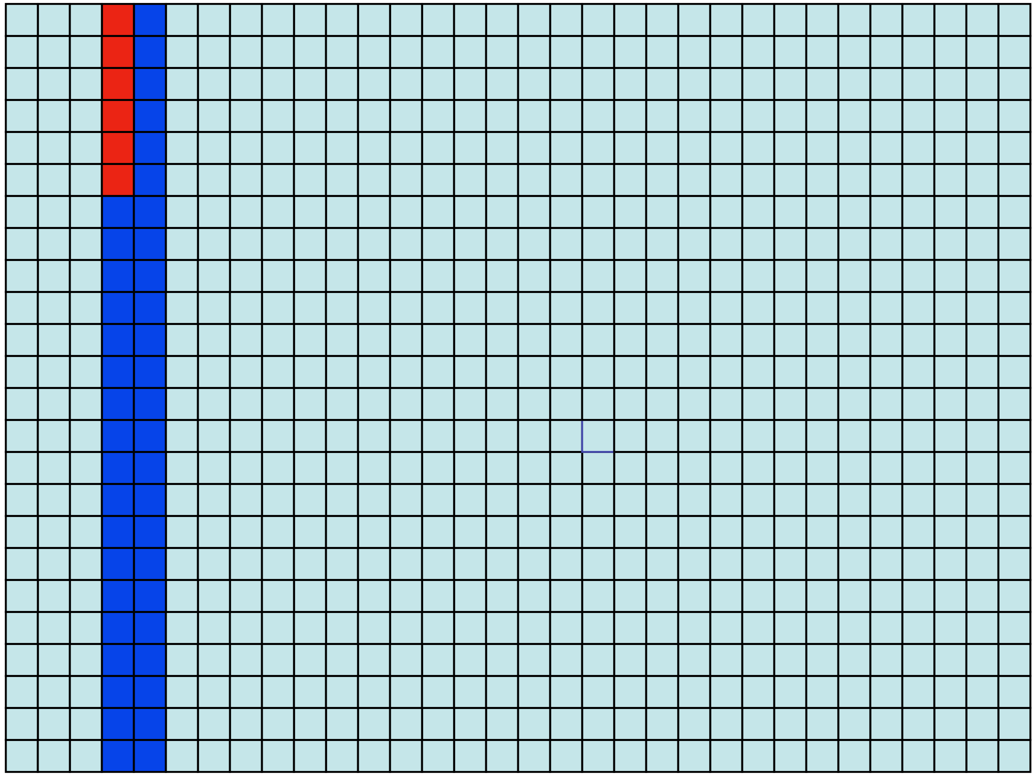


Loop interchange



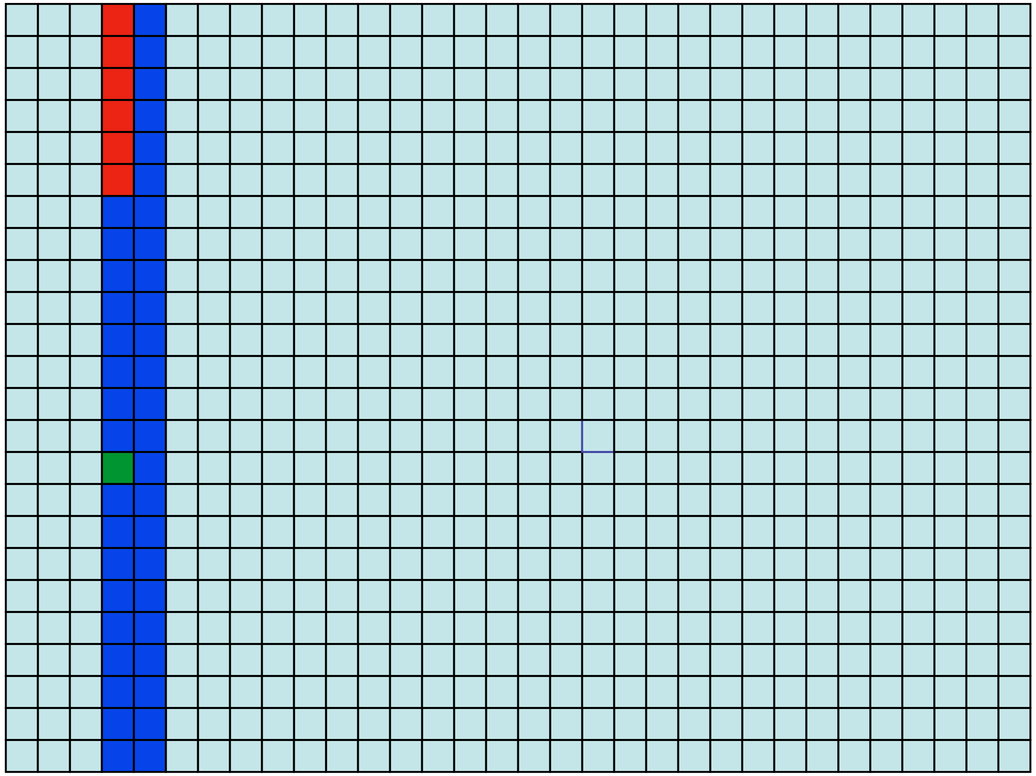
Request first element
In[0][0]

Loop interchange



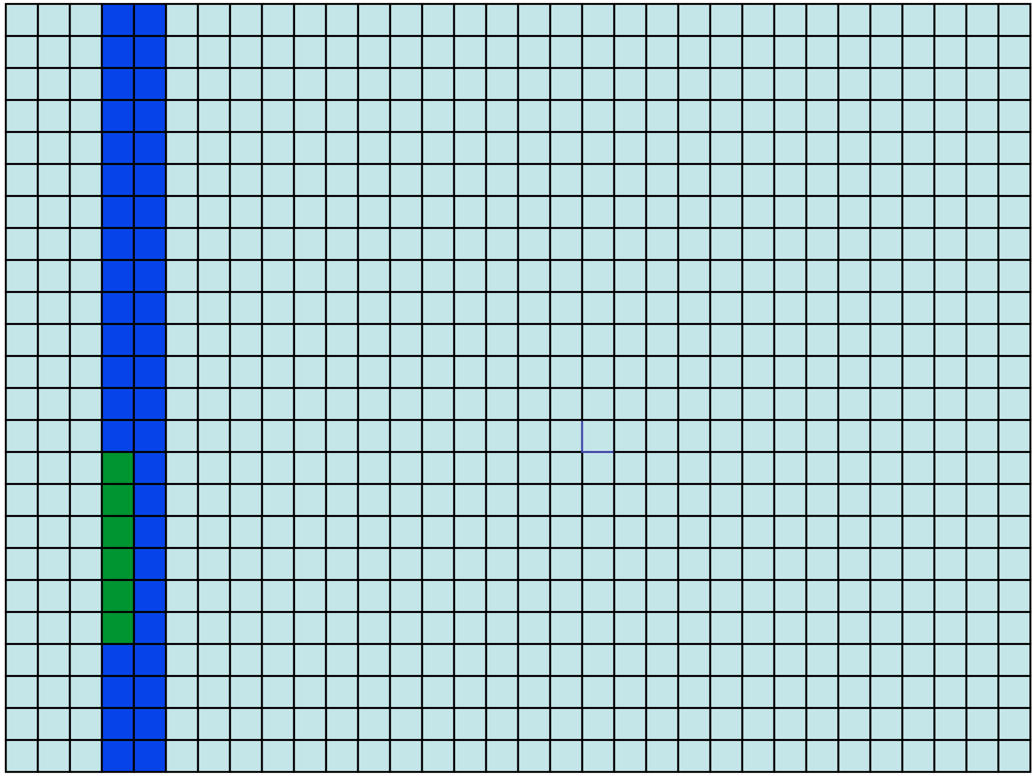
Grab cache line

Loop interchange



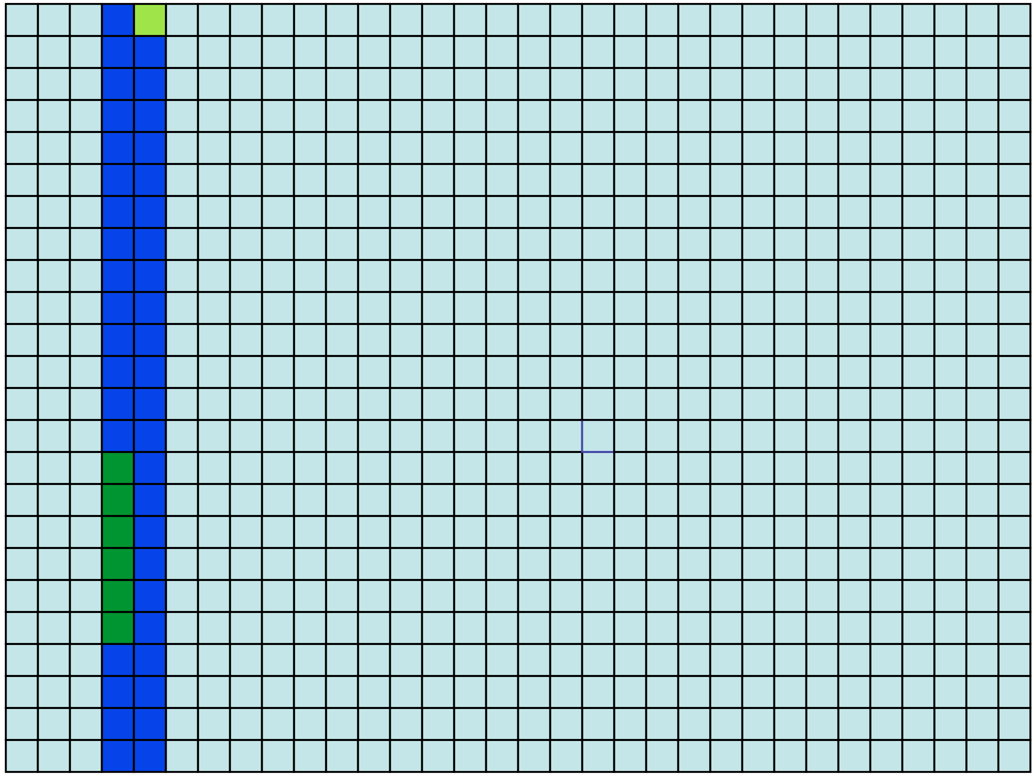
Request in[1][0]

Loop interchange



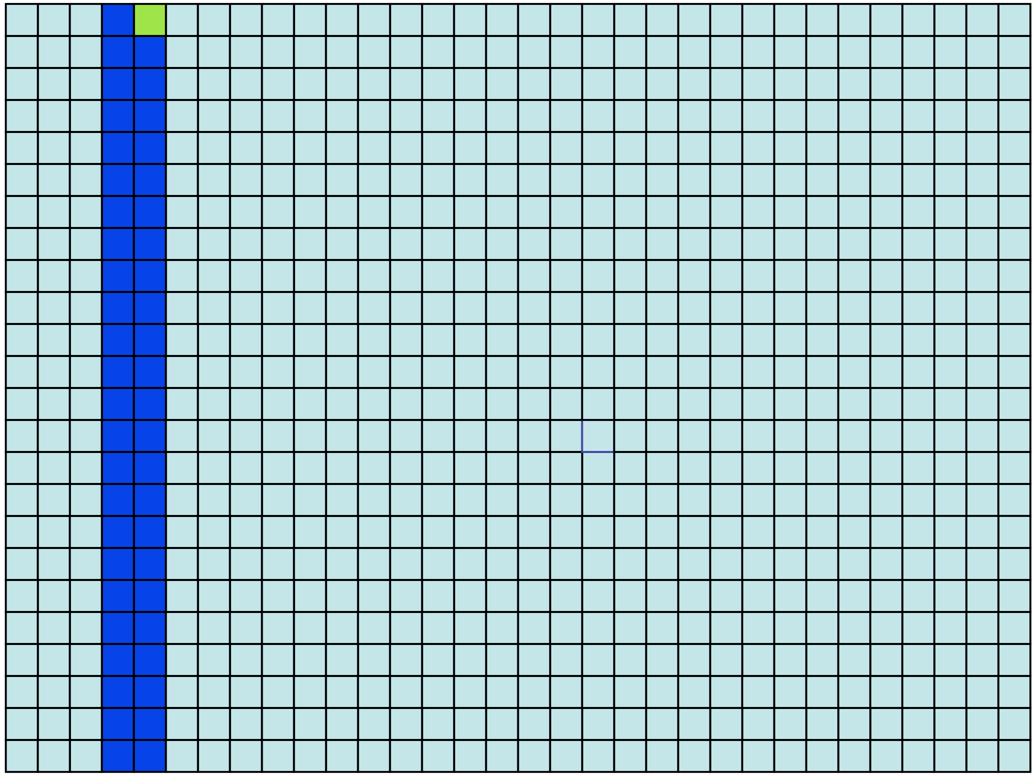
Request in[1][0]
Flush first cache line
Grab new cache line

Loop interchange



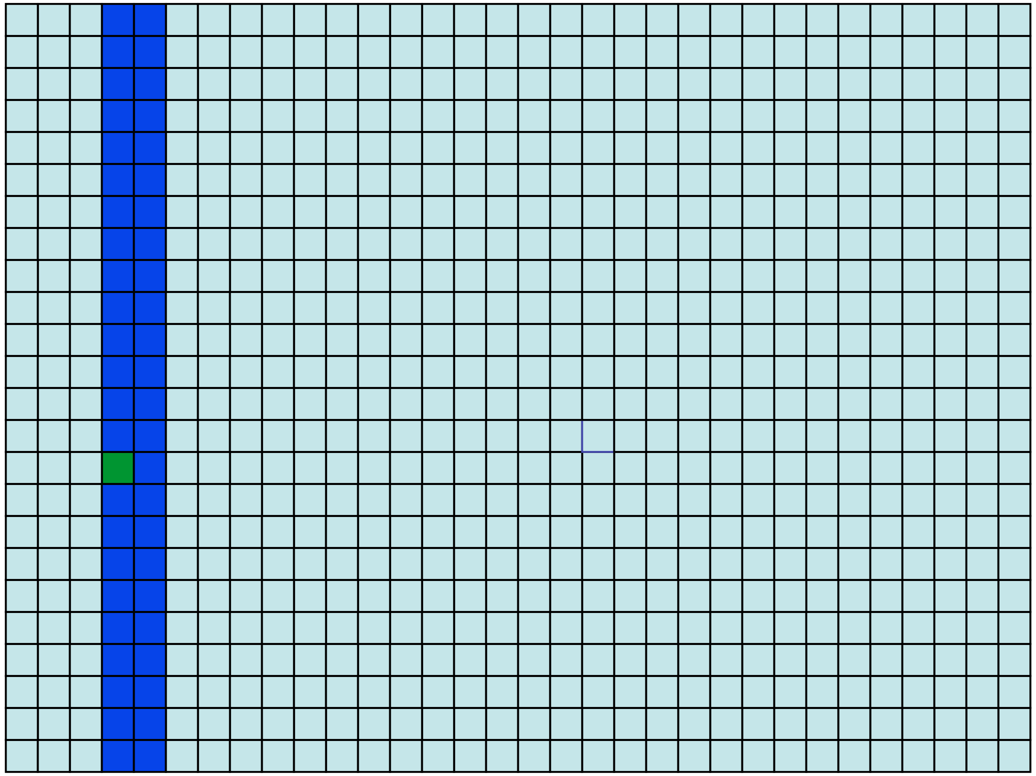
Request in[2][0]

Loop interchange



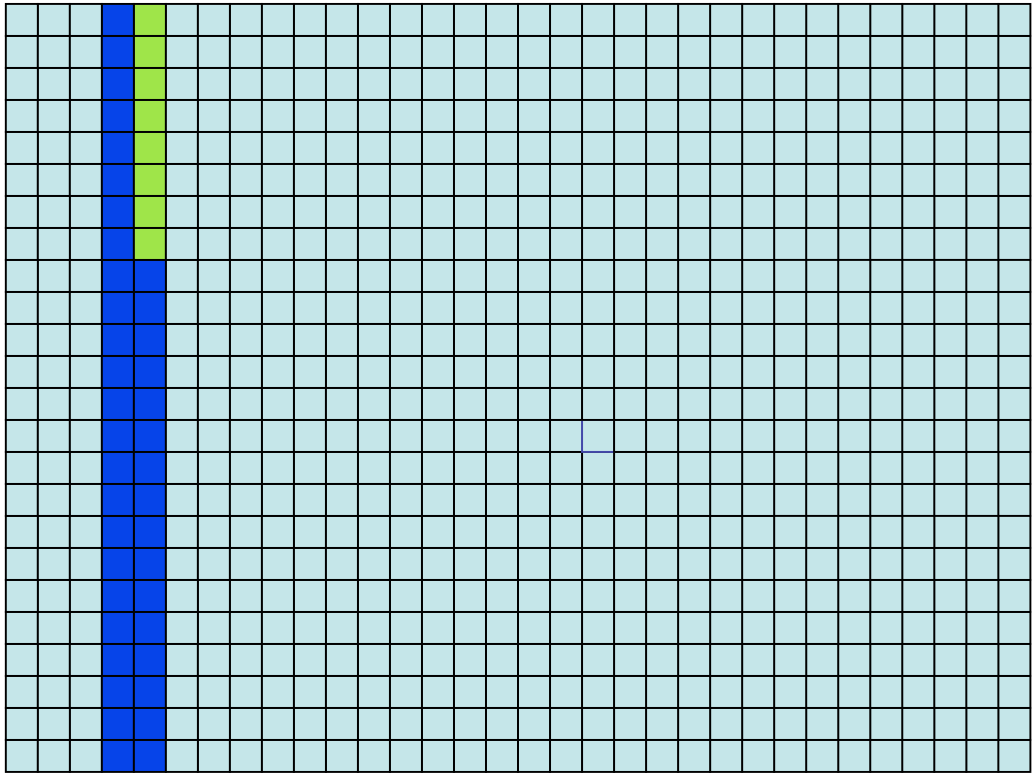
Request in[2][0]
Flush cache line

Loop interchange



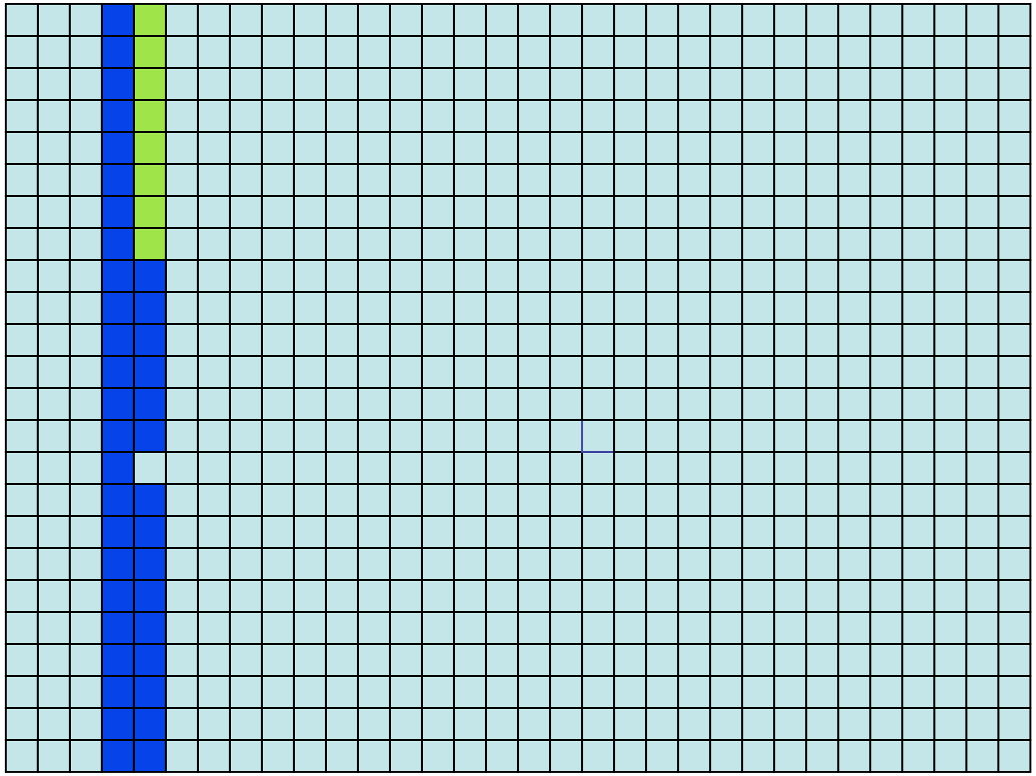
Request in[1][0]
Flush first cache line

Loop interchange



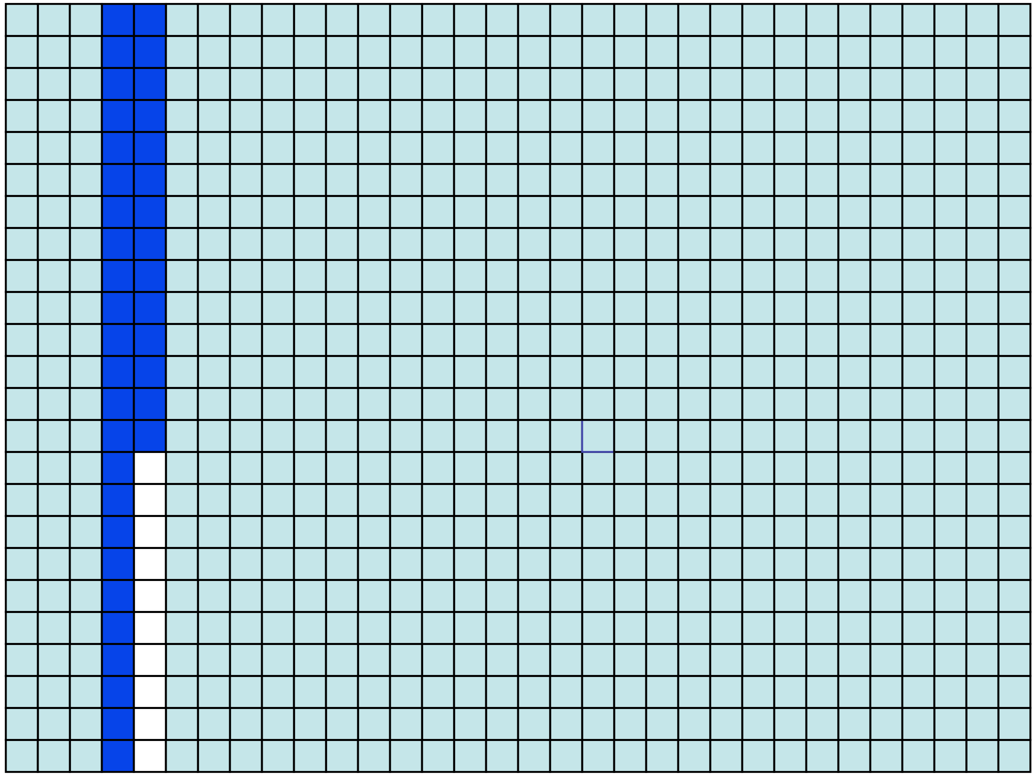
Request in[2][0]
Flush cache line
Grab new cache line

Loop interchange



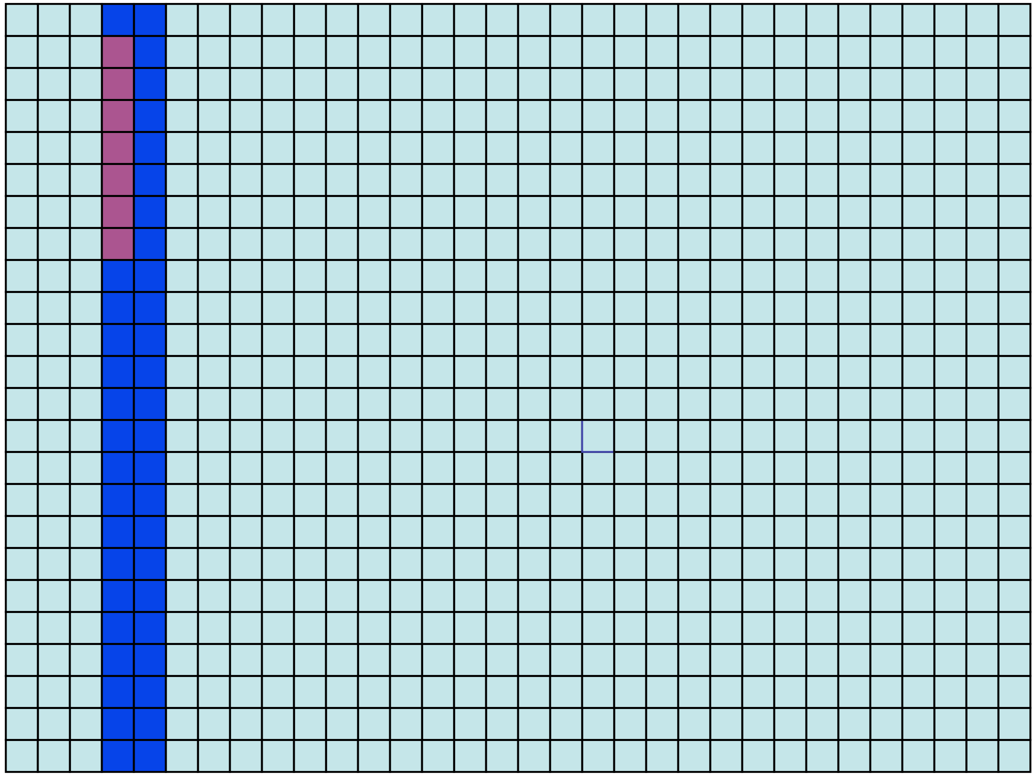
Request in[3][0]

Loop interchange



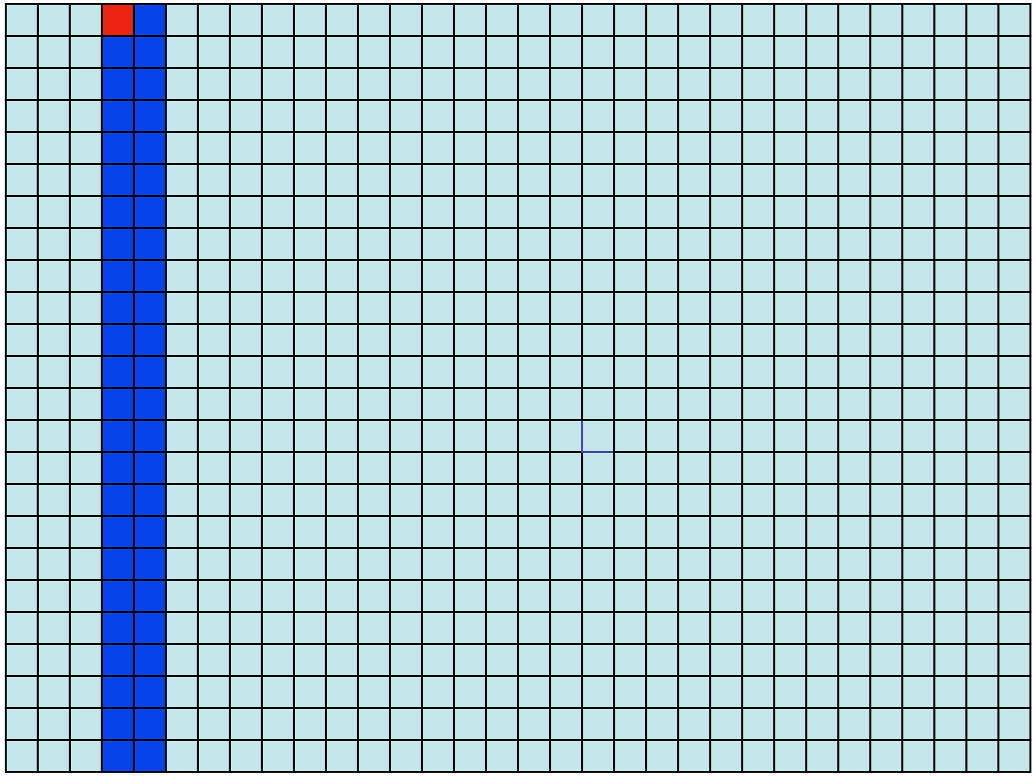
Request in[3][0]
Flush cache line
Grab cache line

Loop interchange



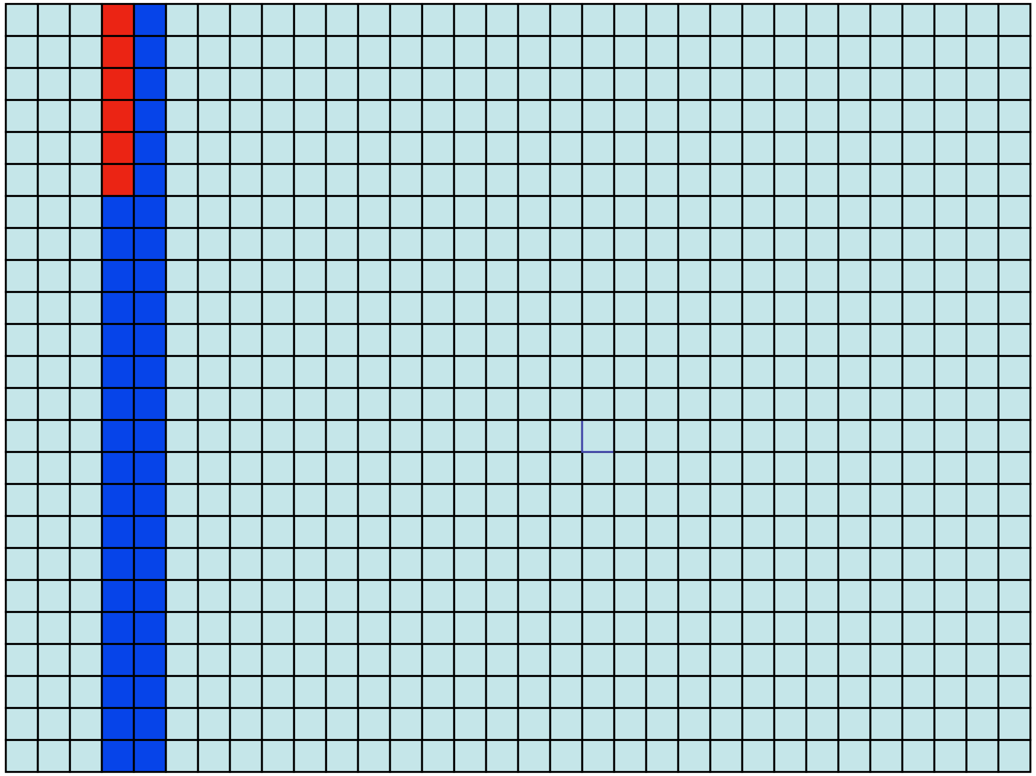
Request in[0][1]
Flush cache line
Grab cache line

Loop reversal



Request first element
In[0][0]

Loop reversal



Request in[0][0]
Grab cache line
Request in[0][1]
Request in[0][2]
Request in[0][3]
.
.
.

Vector operations

```
for(I=0; I < n; I++){
  out[I]=in[I]*in2[I];
}
```

ORIGINAL

```
Vsmult(n,in,in2,out)
```

Vector operations

Most compilers have CPU specific vector routines that substantially improve performance

Loop unswitching

```
for(I=0; I < n; I++){
  if(adj==0) out[I]=in[I]*in2[I];
  else in[I]=out[I]*in2[I];
}
```

ORIGINAL

```
if(adj==0){
  for(I=0; I < n; I++)
    out[I]=in[I]*in2[I];
}
 else{
  for(I=0; I < n; I++)
    in[I]=out[I]*in2[I];
}
```

Removing conditional from loops can substantially improve speed

Example

```
 do imx=down%ax%b, down%ax%n+down%ax%b-1
   jxd = imx - jhx
   jxu = imx + jhx
   if ( (jxd.lt.1) .or. (jxd.gt.size(wfld_d,1)) .or. &
   (jxu.lt.1) .or. (jxu.gt.size(wfld_u,1)) )cycle
      dsliceR(imx-down%ax%b+1, imy-down%ay%b+1, ihx, ihy,ith) =&
      dsliceR(imx-down%ax%b+1, imy-down%ay%b+1, ihx, ihy,ith) +&
       wfld_d(jxd, jyd, iws) * wfld_u(jxu, jyu, iws)
end do
```

Note the if conditional

20x speedup

```
do imx=down%ax%b, down%ax%n+down%ax%b-1
        dsliceR(imx-down%ax%b+1, imy-down%ay%b+1, ihx, ihy,ith) = &
         & dsliceR(imx-down%ax%b+1, imy-down%ay%b+1, ihx, ihy,ith) + &
         & wfld_d(jjxd, jjyd, iws) * wfld_u(jjxu, jjyu, iws)
    end do
```

Loop nest optimizations

```
do I=1,n; do j=1,n;c(I,j)=0
      do k=1,n
           c(I,j)=a(I,k)*b(k,j)
       end do
   end do ;end do
```

ORIGINAL

```
do I=1,n,2; do j=1,n,2
 a01=0;a11=0;a10=0.;a=0;
  do k=1,n
    a00=a(I,k)*b(k,j);a01=a(I,k)*b(k,j+1)
    a10=a(I+1,k)*b(k,j);a11=a(I+1,k)*b(k,j+1)
   end do
  c(I:I+1,j)=(/a00,a01/)
  c(I+1,j:j+1)=(/a10,a11/)
 end do ;end do
```

Each input array element is used twice per read

Loop invariant code motion

```
do while(j< sum(array))
  j=j+sqrt(alpha*alpha+beta*beta)
end do
```

Original

```
m=sum(array)
n=sqrt(alpha*alpha+beta*beta)
do while(j < m)
  j=j+n
end do
```

Precompute variables that are unchanging in a loop

Functional inlining

```
real :: array(n1),in(n1)
integer :: n1
integer :: i1
        do i1=1,n1
              array(i1)=mypow(in(i1))
         end do
```

```
real :: array(n1),in(n1)
integer :: n1
integer :: i1
        do i1=1,n1
              array(i1)= in(i1)**2
         end do
```

```
real function mypow(var)
  real :: var
  mypow=var**2
```

$\underbrace{\hspace{6em}}_{\textbf{100 times faster}}$

100 times faster

Machine-specific optimization

- Many choices the compiler makes can be improved by knowing more about the target CPU
  - Number of registers
  - Number of floating point units
  - Cache
- For highly optimized but non-portable code, turn on these machine-specific optimizations
- The speed difference is often a factor of 2 or better

Relative importance

- Loop unrolling
- Loop interchange
- Vectorizing operations
- Loop unswitching
- Loop nest optimizations
- Loop invariant code motion
- **Inlining**
- Machine-specific optimization

To important to trust to the compiler

Worth it in many cases

Not worth you doing, trust to the compiler, but make his job doable

Making the compiler work for YOU

- Simple code
- Include all portions of an expensive section of your project in one file
- Do not use fancy pointer tricks
- Don’t do fancy self-optimizations

Example 1: Vectorizing operations

```
 do ihy=1,size(wfld,4)  ; do ihx=1,size(wfld,3)
     do imy=1,wsep%n(2); do imx=1,wsep%n(1)
             k = sqrt(kx(imx,ihx)**2 + ky(imy,ihy)**2)
             i =max(1, min(int(1 + k/ko / dkxko) , nkxko))
             ikz= ko*tkzko(i) *dstep
             wfld(imx,imy,ihx,ihy,iws) = &
               wfld(imx,imy,ihx,ihy,iws) * cexp( ikz * dstep)
end do;end do;end do; end do
```

Example 1: Vectorizing operations

Precomputed variables

```
do ihy=1,size(wfld,4)  ; do ihx=1,size(wfld,3)
    do imy=1,wsep%n(2); do imx=1,wsep%n(1)
            k = sqrt(kx(imx,ihx)**2 + ky(imy,ihy)**2)
            i =max(1, min(int(1 + k/ko / dkxko) , nkxko))
            ikz= ko*tkzko(i) *dstep
            wfld(imx,imy,ihx,ihy,iws) = &
              wfld(imx,imy,ihx,ihy,iws) * cexp( ikz * dstep)
end do;end do;end do; end do
```

Example 1: Vectorizing operations

```
do ihy=1,size(wfld,4)  ; do ihx=1,size(wfld,3)
    do imy=1,wsep%n(2); do imx=1,wsep%n(1)
        k = sqrt(kx(imx,ihx)**2 + ky(imy,ihy)**2)
        i =max(1, min(int(1 + k/ko / dkxko) , nkxko))
        ikz= ko*tkzko(i) *dstep
        wfld(imx,imy,ihx,ihy,iws) = &
          wfld(imx,imy,ihx,ihy,iws) * cexp( ikz * dstep)
end do;end do;end do; end do
```

Costly operation

Example 1: Vectorizing operations

```
do ihy=1,size(wfld,4)  ; do ihx=1,size(wfld,3)
   do imy=1,wsep%n(2); do imx=1,wsep%n(1)
         k = sqrt(kx(imx,ihx)**2 + ky(imy,ihy)**2)
         i =max(1, min(int(1 + k/ko / dkxko) , nkxko))
         ikz= ko*tkzko(i) *dstep
         wfld(imx,imy,ihx,ihy,iws) = &
          wfld(imx,imy,ihx,ihy,iws) * cexp( ikz * dstep)
end do;end do;end do; end do
```

$$e^{a+ib} = e^{a}(\cos(b) + j\sin(b))$$

Example 1: Vectorizing operations

```
do ihy=1,size(wfld,4)  ; do ihx=1,size(wfld,3)
    do imy=1,wsep%n(2); do imx=1,wsep%n(1)
            k = sqrt(kx(imx,ihx)**2 + ky(imy,ihy)**2)
            i =max(1, min(int(1 + k/ko / dkxko) , nkxko))
            ikz= ko*tkzko(i) *dstep
            sc(imx)=cmplx(cos(aimag(ikz)),sin(aimag(ikz)))
            bout(imx)=real(ikz)
         end do
         bout=exp(bout)
         wfld(:,imy,ihx,ihy,iws) = &
            wfld(:,imy,ihx,ihy,iws) * dstep*bout*sc
end do;end do;end do
```

Example 1: Vectorizing operations

```
do ihy=1,size(wfld,4)  ; do ihx=1,size(wfld,3)
    do imy=1,wsep%n(2); do imx=1,wsep%n(1)
        k = sqrt(kx(imx,ihx)**2 + ky(imy,ihy)**2)
        i =max(1, min(int(1 + k/ko / dkxko) , nkxko))
        ikz= ko*tkzko(i) *dstep
        sc(imx)=cmplx(cos(aimag(ikz)),sin(aimag(ikz)))
        bout(imx)=real(ikz)
      end do
      bout=exp(bout)
      wfld(:,imy,ihx,ihy,iws) = &
        wfld(:,imy,ihx,ihy,iws) * dstep*bout*sc
end do;end do;end do
```

Factor 10 speed up

Example 2: Loop invariant code motion

```
do isx=1,nsx_in,nsrx_in
    do isy=1,nsy_in,nsry_in
        pp=aaai(isy,isx)*aaaj(isy,isx)
         do irx=1,nsx_in
         do iry=1,nsy_in
             ppp=ppp+real(pp*aaai(iry,irx)*aaaj(iry,irx))
         enddo
         enddo
     enddo
enddo
```

Example 2: Loop invariant code motion

```
 do isx=1,nsx_in,nsrx_in
    do isy=1,nsy_in,nsry_in
        pp=aaai(isy,isx)*aaaj(isy,isx)
         do irx=1,nsx_in
         do iry=1,nsy_in
             ppp=ppp+real(pp*aaai(iry,irx)*aaaj(iry,irx))
         enddo
         enddo
     enddo
enddo
```

Example 2: Loop invariant code motion

```
mult=aaai*aaaj
mys=sum(mult)
do isx=1,nsx_in,nsrx_in
    do isy=1,nsy_in,nsry_in
        ppp=real(mys*mult(isx,isy))
    enddo
enddo
```

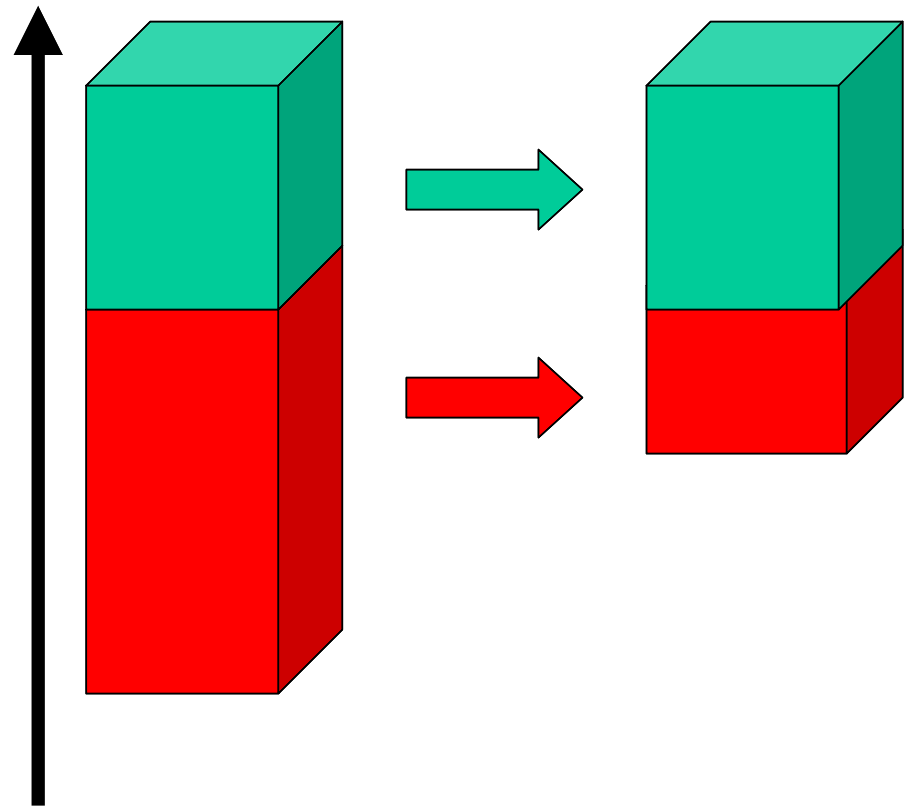
Factor 50 speedup

Diminishing Returns

Improvement is never as good as you might expect. For example,

Red routine runtime improved by factor of 3

But total runtime improved only by factor of 1.5



What can be tuned?

Execution time $T = Ti + Td$

Ti = Time to execute instructions

Td = Time to move data in and out of processor (expensive)

- $Ti = (\Sigma \text{ instructions}) \times (\text{time/instruction})$
- $Td = (\Sigma \text{ memory ops}) \times (\text{time/memop})$

All these four components may be improved.

What is wrong with this code?

Assuming
n1*n2>L2 size

```
real :: array(n1,n2,nt),in(n1,n2,nt)
real :: o1,o2,d1,d2
integer :: n1,n2,nt
integer :: i1,i2,iter
do iter=1,nt
  do i1=1,n1
    do i2=1,n2
       out(i1,i2,iter)=iter*in(i1,i2,iter)*cos(o1+d1*(i1-1))*sin(o2+d2*(i2-1))
   end do
  end do
end do
```

What <u>not</u> to do when looping: Fortran

Assuming
n1*n2>L2 size

```
real :: array(n1,n2,nt),in(n1,n2,nt)
real :: o1,o2,d1,d2
integer :: n1,n2,nt
integer :: i1,i2,iter
do iter=1,nt
  do i1=1,n1
    do i2=1,n2
       out(i1,i2,iter)=iter*in(i1,i2,iter)*cos(o1+d1*(i1-1))*sin(o2+d2*(i2-1))
    end do
  end do
end do
```

Continual cache misses
Because of loop order

72 seconds

Correct looping in Fortran

```
real :: array(n1,n2,nt),in(n1,n2,nt)
real :: o1,o2,d1,d2
integer :: n1,n2,nt
integer :: i1,i2,iter
do iter=1,nt
  do i2=1,n2
    do i1=1,n1
       out(i1,i2,iter)=iter*in(i1,i2,iter)*cos(o1+d1*(i1-1))*sin(o2+d2*(i2-1))
   end do
  end do
end do
```

Better use of cache

23 seconds

What not to do looping: C

Note the reversal in loop order

```
float ***in, ***out;
float o1,o2,d1,d2;
int  n1,n2;
int  i1,i2,it;
  for(it=0; it< niter; it++){
    for(i2=0; i2 < n2; i2++){
      for(i1=0;i1 < n1; i1++){
        out[it][i2][i1]=iter*in[it][i2][i1]*cos(o1+d1*i1)*sin(o2+d2*i2);
      }
    }
  }
```

Remove constants from the inner loop

```
real :: array(n1,n2,nt),in(n1,n2,nt)
real :: o1,o2,d1,d2,y
integer :: n1,n2
integer :: i1,i2,iter
do iter=1,nt
  do i2=1,n2
   y=sin(o2+d2*(i2-1))
    do i1=1,n1
       out(i1,i2,iter)=iter*in(i1,i2,iter)*cos(o1+d1*(i1-1))*y
    end do
  end do
end do
```

sin unchanging inner loop

15 seconds

Remove constants from the inner loop

```
real :: array(n1,n2,nt),in(n1,n2,nt)
real :: o1,o2,d1,d2,x(n1),y(n2)
integer :: n1,n2
integer :: i1,i2,iter
do i2=1,n2; y(i2)=sin(o2+d2*(i2-1));end do
do i1=1,n1; x(i1)=cos(o1+d1*(i1-1));end do
do iter=1,nt
  do i2=1,n2
    do i1=1,n1
      out(i1,i2,iter)=iter*in(i1,i2,iter)*y(i2)*x(i1)
    end do
  end do
end do
```

Precalculate both cos and sin

5 seconds

Relative cost of operations

Operation	**Cost**
Addition	1
Subtraction	1
Multiplication	2
Division	4
Exponential	5
Trigonomic function	7
Complex exponential	18

How much precision do you need?

- 4 byte arithmetic (float or real) is almost twice as fast as 8 byte arithmetic (double or double precision)
- A table lookup often is even a better option when precision isn’t essential and the operation is expensive

Table lookups

```
subroutine calc_cos(input,output,n,o,d)
  integer :: i,index,n
  real      :: o,d, input(:),output(:),v
  real,allocatable:: cos_look(:)

  allocate(cos_look(n))
  do i=1,n
    v=o+d*(i-1)
    cos_look(i)=cos(v)
  end do
  do i=1,size(input)
    index=(input(i)-o)/d+1.5
    output(i)=cos_look(index)
  end do
end subroutine
```

If you are doing a costly operation, and a high level precision isn't required, it can be advantageous to use table lookups.

With modern compilers, CPUs, inner operator must take at least 10 clock cycles

Table lookups

```
do ihy=1,size(wfld,4)  ; do ihx=1,size(wfld,3)
   do imy=1,wsep%n(2); do imx=1,wsep%n(1)
         k = sqrt(kx(imx,ihx)**2 + ky(imy,ihy)**2)
         if(k < 1.) then
           ikz=cmplx(0.,sqrt(1.-k**2)
         else
           ikz=cmplx(sqrt(k**2-1.),0.)
         end if
         wfld(imx,imy,ihx,ihy,iws) = &
          wfld(imx,imy,ihx,ihy,iws) * cexp( ikz * dstep)
end do;end do;end do; end do
```

Table lookups

```
do ihy=1,size(wfld,4)  ; do ihx=1,size(wfld,3)
    do imy=1,wsep%n(2); do imx=1,wsep%n(1)
        k = sqrt(kx(imx,ihx)**2 + ky(imy,ihy)**2)
        i =max(1, min(int(1 + k/ko / dkxko) , nkxko))
        ikz= ko*tkzko(i) *dstep
        wfld(imx,imy,ihx,ihy,iws) = &
         wfld(imx,imy,ihx,ihy,iws) * cexp( ikz * dstep)
end do;end do;end do; end do
```

Table lookups

```
    !!                    ____________
!!   kz         1    /    | kx |2          kx
!! i -- dz = -i -  /  1 - | -- |  dz   0 < -- < 1
!!   ko         2 V      | ko |           ko
!!
do i=1,m
   kxko    = 0. +     1.0 * (i  -1)/(  m-1) !! kx/ko=0...1
   kzko(i) = - (0,+1) * sqrt(1-kxko**2)
end do

!!                   ____________
!!   kz         1      / | kx |2             kx
!! i -- dz = -  -   /  | -- | - 1  dz   1 < -- < max
!!   ko         2 V   | ko |              ko
!!
do i=m+1,n
   kxko    = 1. + (max-1) * (i-m-1)/(n-m-1) !! kx/ko=1...max
   kzko(i) = - (+1,0) * sqrt(kxko**2-1)
end do
```

Some simple speed comparisons

- We are going to compare how to write a simple function in several different ways without any compiler optimization
- For this simple function the compiler can successfully optimize all of the variations
- The goal is to demonstrate some basic guidelines to make the compiler's job as easy as possible

A simple speed test

```
do itime=1,ntimes
  do i3=1,n3
  do i2=1,n2
  do i1=1,n1
    buf2(i1,i2,i3)=buf(i1,i2,i3)+3.4
  end do
  end do
  end do
end do
```

N1=100,n2=1000
n3=1000,ntimes=10

14.4 seconds

A simple speed test: -O3

```
do itime=1,ntimes
  do i3=1,n3
  do i2=1,n2
  do i1=1,n1
    buf2(i1,i2,i3)=buf(i1,i2,i3)+3.4
  end do
  end do
  end do
end do
```

N1=100,n2=1000
n3=1000,ntimes=10

3.9 seconds

Simple speed test in C

Time ranges from 4.3 to 9 seconds

```
for(itimes=0; itimes < ntimes; itimes++){
for(i3=0; i3< n3; i3++){
for(i2=0; i2< n2; i2++){
for(i1=0; i1< n1; i1++){
ar2[i3][i2][i1]=ar1[i3][i2][i1]+3.4;
}}}
}
```

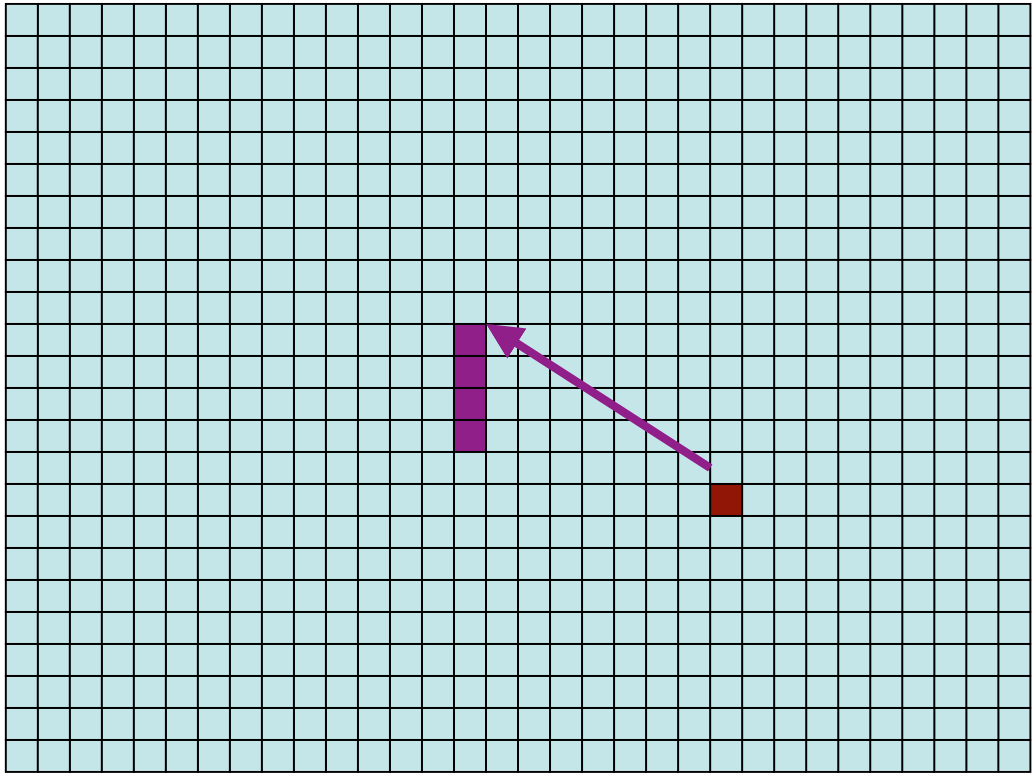
Difference is based on allocation methodology

Simple speed test in C: Allocating non-continuous memory

```
ar1=(float ***)malloc(sizeof(float **)*n3);
ar2=(float ***)malloc(sizeof(float **)*n3);
for(i3=0; i3< n3; i3++){
  ar1[i3]=(float **)malloc(sizeof(float *)*n2);
  ar2[i3]=(float **)malloc(sizeof(float *)*n2);
}
for(i3=0; i3< n3; i3++){
for(i2=0; i2< n2; i2++){
  ar1[i3][i2]=(float *)malloc(sizeof(float )*n1);
  ar2[i3][i2]=(float *)malloc(sizeof(float )*n1);
}}
```

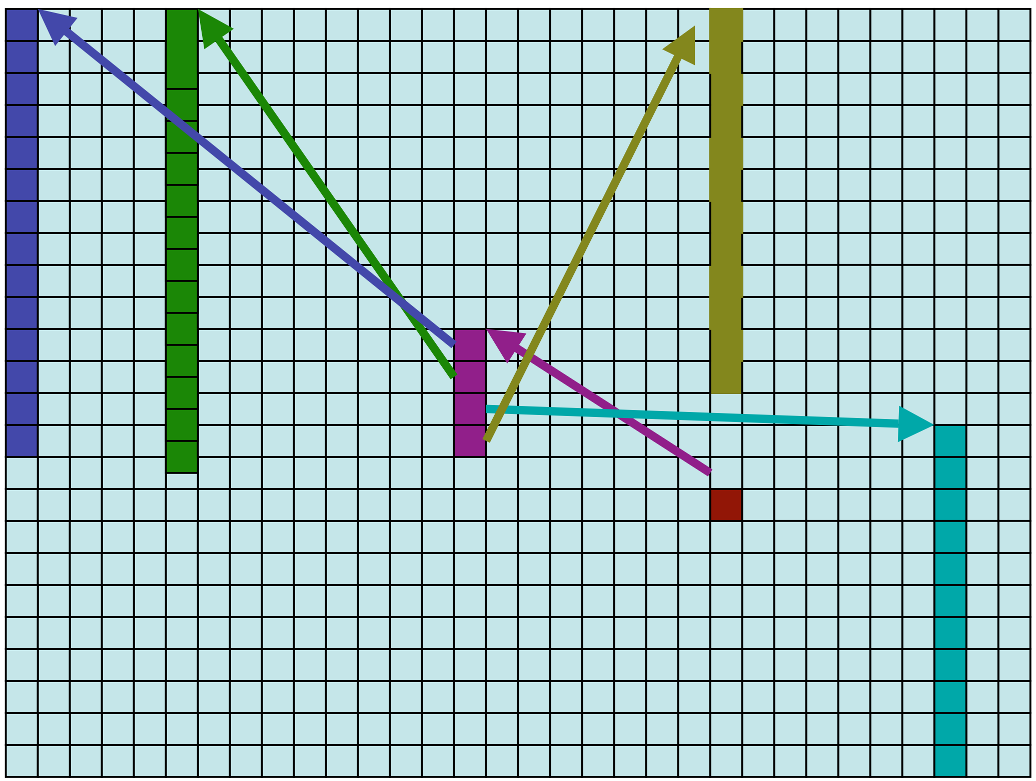
Time ranges from 5.3 seconds

Pointer: 2-D array



Allocate a pointer to an array of pointers

Pointer: 2-D array



A batch of memory is allocated and the pointer is set to the memory location of the first element

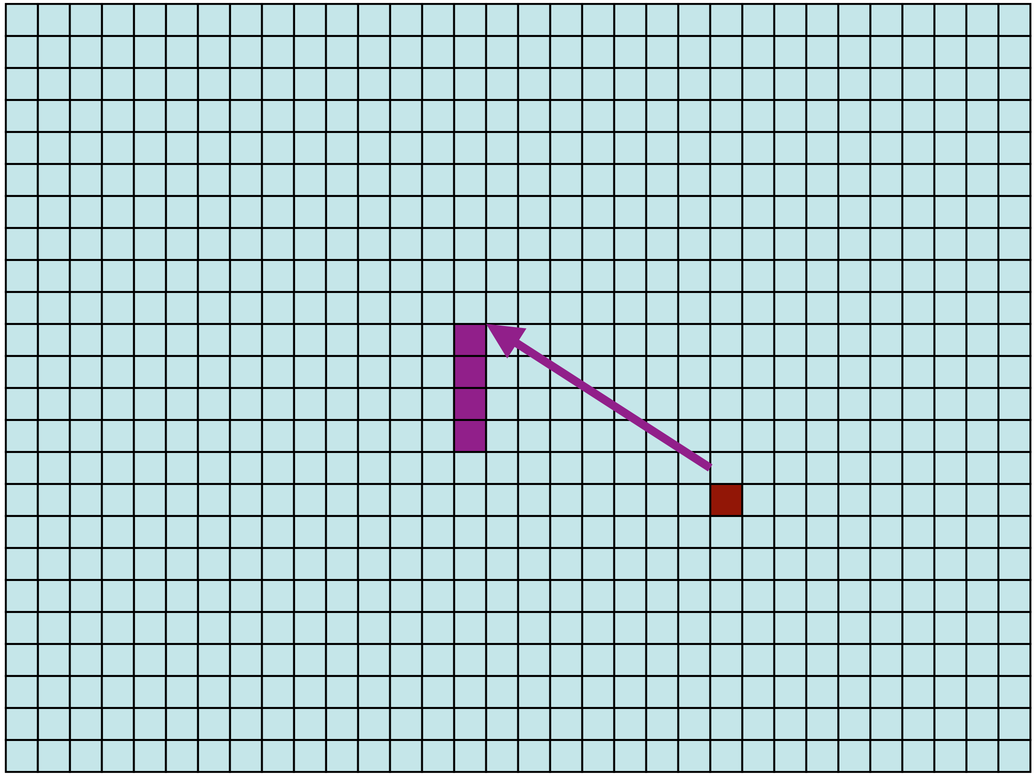
In this case each element is another memory location pointer

Simple speed test in C: Allocating continuous memory

```
 ar1=(float***)malloc(n3*sizeof(float**));
ar1[0]=(float**)malloc(n3*n2*sizeof(float*));
ar1[0][0]=(float*)malloc(n3*n2*n1*size));
for (i3=0; i3<n3; i3++) {
   ar1[i3] = ar1[0]+n2*i3;
    for (i2=0; i2<n2; i2++) ar1[i3][i2] =
         (float*)ar1[0][0]+size*n1*(i2+n2*i3);
     }
}
```

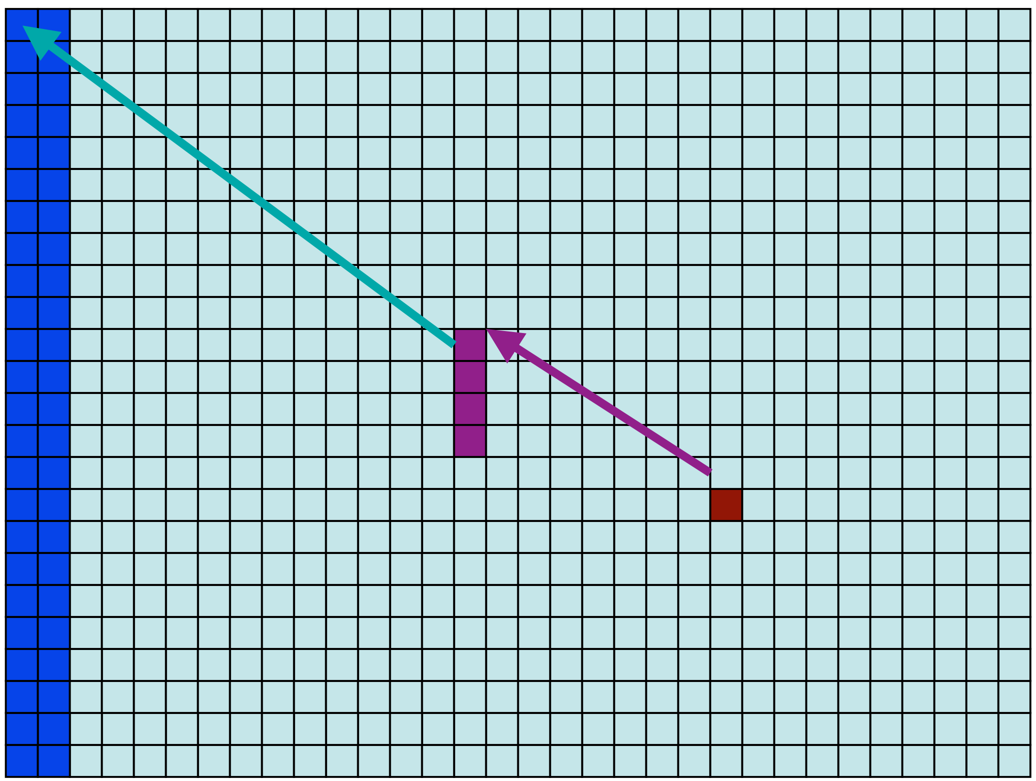
Time ranges from 4.7 -9.5 seconds

Pointer: 2-D array



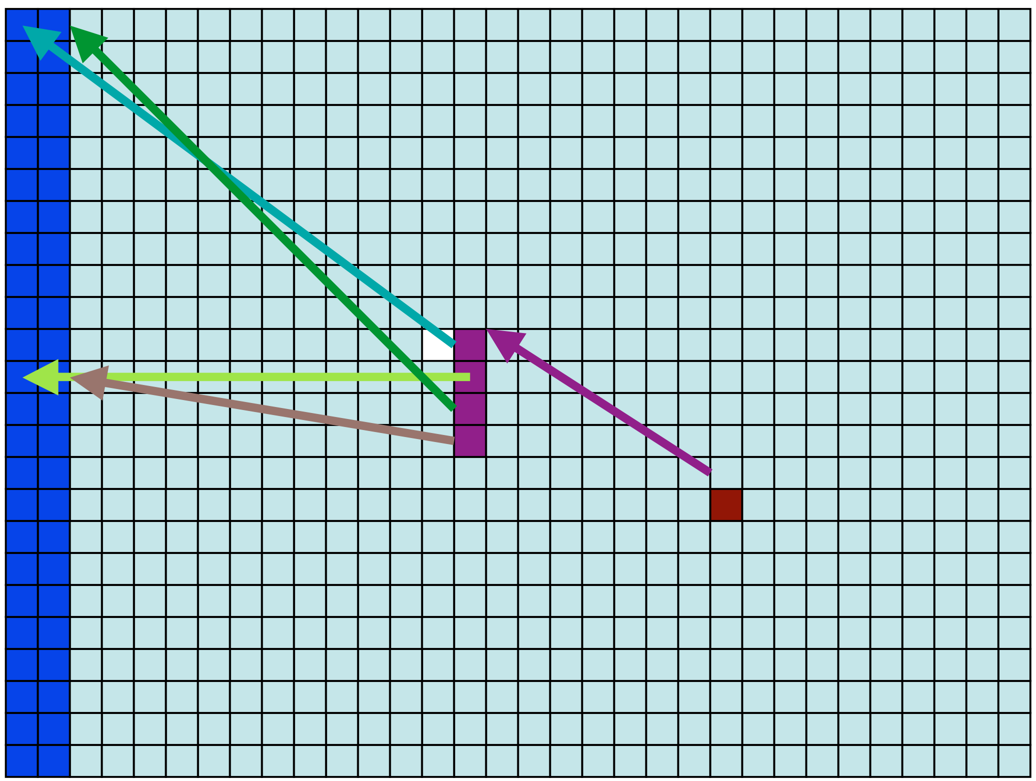
Allocate a pointer to an array of pointers

Pointer: 2-D array



Allocate the entire block to the first element of the pointer array

Pointer: 2-D array



Point the remaining elements inside the allocated block

Allows for contiguous memory

You can do operations that are not dependent dimensionality by referencing array[0]

Simple speed test in C: If you put the allocation in another file

```
ar1=(float***)malloc(n3*sizeof(float**));
ar1[0]=(float**)malloc(n3*n2*sizeof(float*));
ar1[0][0]=(float*)malloc(n3*n2*n1*size));
for (i3=0; i3<n3; i3++) {
   ar1[i3] = ar1[0]+n2*i3;
    for (i2=0; i2<n2; i2++) ar1[i3][i2] =
         (float*)ar1[0][0]+size*n1*(i2+n2*i3);
     }
}
```

Time ranges from 9.5 seconds

Fortran arrays

- You can use either the pointer or allocatable attribute for an array
- The same problem that makes Fortran generally faster than C can make pointer arrays slower than allocatable

Fortran array types

- Automatic arrays
  - real :: array(n1,n2)
- Allocatable arrays
  - real, allocatable :: array(:,:)
- Pointer arrays
  - real, pointer :: array(:,:)

Automatic arrays

- Automatic arrays are created on the stack when entering a new functional unit
- They are automatically destroyed when exiting the functional unit
- For relatively small arrays, this allows the compiler the most flexibility in optimization

Heap memory

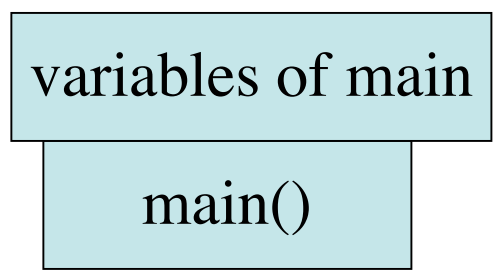
- All global variables (static) live on the heap
- In heap-based memory allocation, memory is allocated from a large pool of unused memory area called the heap. The size of the memory allocation can be determined at run-time, and the lifetime of the allocation is not dependent on the current procedure or stack frame. The region of allocated memory is accessed indirectly, usually via a reference.- Wikipedia
- Generally the heap isn't as efficient as the stack but offers much more programming flexibility

Stack allocation

main()

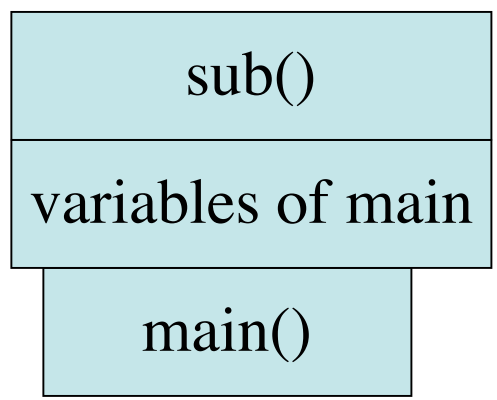
Top of the stack contains
the main program

Stack allocation



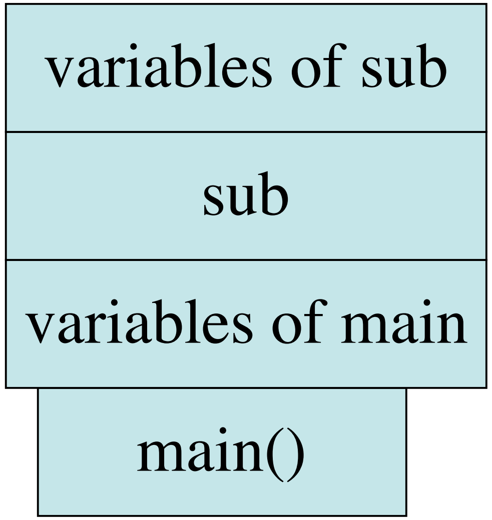
Variables declared in main are put on top of the stack

Stack allocation



When a subroutine is called it is placed on top of the stack

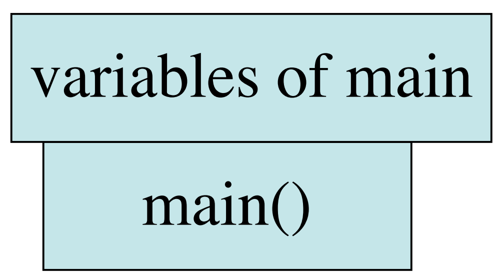
Stack allocation



Variables in the subroutine are then placed on top of the stack

Stack allocation

When the subroutine is exited, its contents are flushed from the stack



Stack allocation

- The stack operates on a last-in first-out principal
- Accessing variables on the stack is generally more efficient than accessing on the heap
- The problem is that the size of the stack is limited to a specific size at runtime

Problems with stack allocation 1: Automatic arrays

If the memory the stack needs at any given time exceeds the preset stack memory size you will get a segmentation fault with no diagnostic information. One common way to exceed the stack is by large automatic arrays

```
real :: array(n),array(n)
```

If n is large and/or there are many arrays, or recursive calls to a subroutine with automatic arrays, you are in danger of exceeding the computer's stack limit.

Problems with stack allocation 2: Matrix and vector operations in Fortran

```
subroutine add_matrix(array1,array2,array3)
  real :: array1(:,:),array2(:,:),array3(:,:)
    array3=array1+array2
end subroutine
```

Almost all compilers create a temporary, automatic array to contain the result of array1+array2

Example

```
do isy=1,nsy_in
  do isx=2,nsx_in
    ip =max(1,min(nsux_in,ixrtart+(isx-1)*jsx_in))
    iip=max(1,min(nsux_in,ip+nrx_in))
    jp =max(1,min(nsux_in,ixrtart+isx*jsx_in))
    jjp=max(1,min(nsux_in,jp+nrx_in))
    mysa(isx,isy)=mysa(isx-1,isy)+ sum(aaaii(ip:jp,1)*aaajj(ip:jp,1))+sum(aaaii(iip:jjp,1)*aaajj(iip:jjp,1))
  end do
end do
```

```
do isy=1,nsy_in
   do isx=2,nsx_in
     ip =max(1,min(nsux_in,ixrtart+(isx-1)*jsx_in))
     iip=max(1,min(nsux_in,ip+nrx_in))
     jp =max(1,min(nsux_in,ixrtart+isx*jsx_in))
     jjp=max(1,min(nsux_in,jp+nrx_in))

     mysm=sum(aaaii(ip:jp,1)*aaajj(ip:jp,1))    !Sum(G'(x   ,y   ,z,xr,yr)*G(x+ax,y+ay,z+az,xr,yr))     '
     mysp=sum(aaaii(iip:jjp,1)*aaajj(iip:jjp,1))
     mysa(isx,isy)=mysa(isx-1,isy)+mysp-mysm
   end do
 end do
```

10x speed up

Overloading

- C++ and Fortran allow you to overload basic mathematical operations (+,-),.etc.
  - Don’t do it
- The same memory, speed penalty associated with Fortran automatic arrays

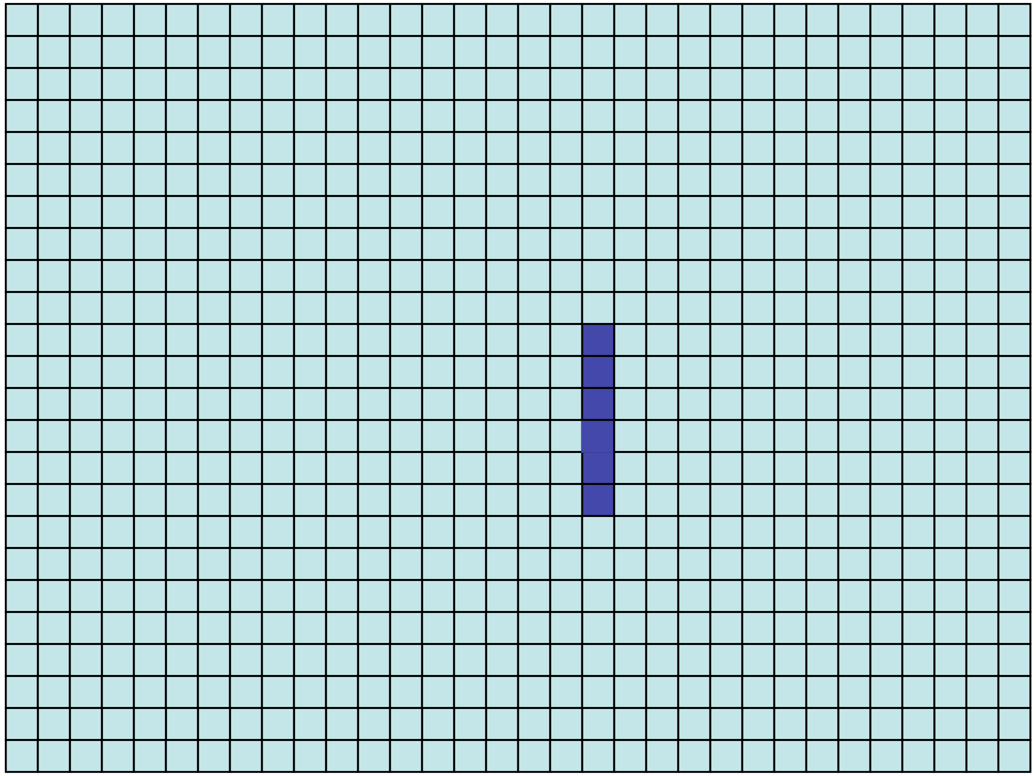
Allocatable arrays

- Are allocated on the heap
- Are guaranteed to have non-overlapping memory
- Have to be allocated in the routine that creates them
- In f90 cannot be part of a structure
- Allow the highest level of optimization by the compiler

Pointer arrays

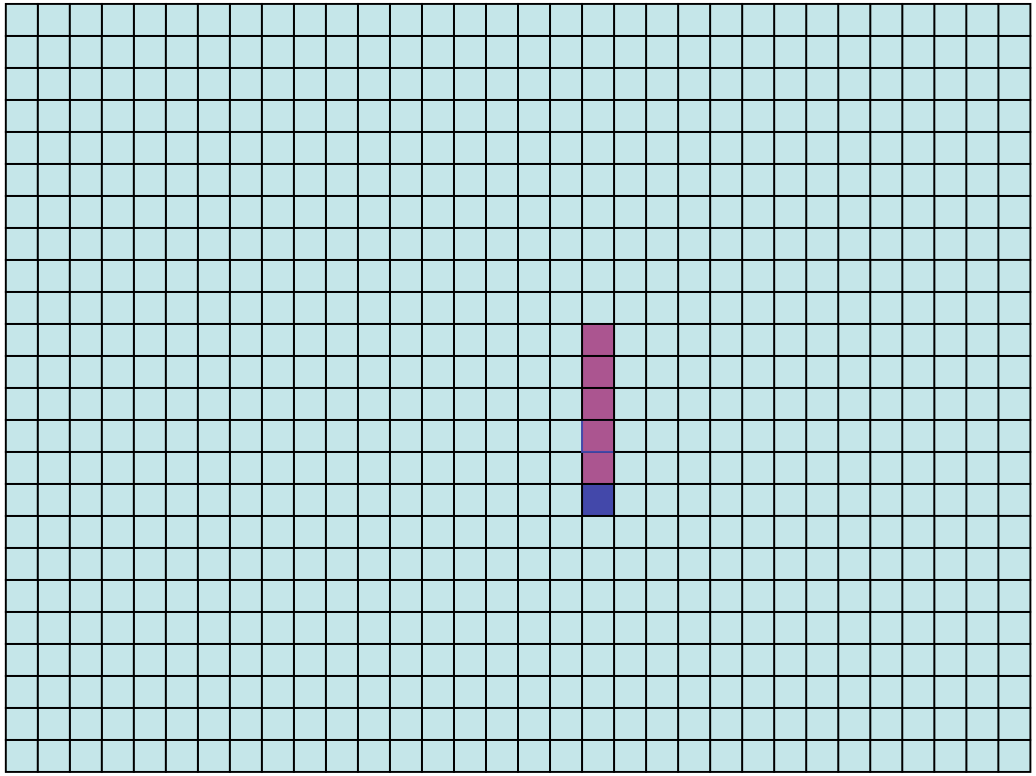
- Are the closest to a pointer in C
- Are allocated in the heap
- Can have overlapping segments
- Are the least efficient of the Fortran array types (but still more efficient then C)

Overlapping memory



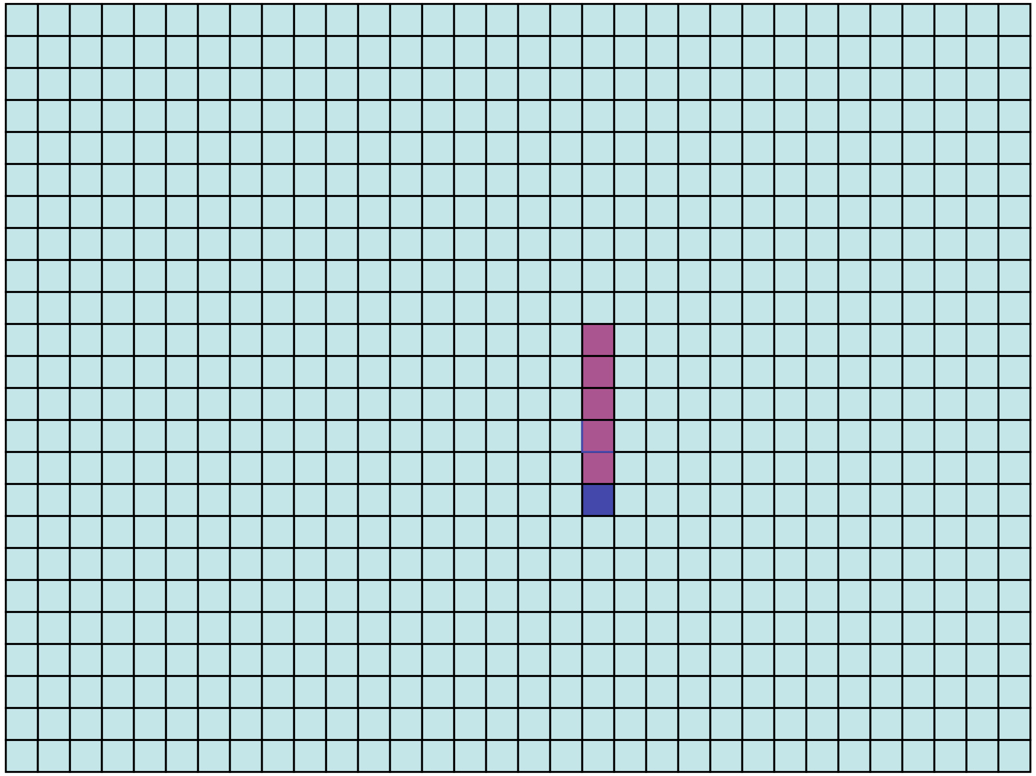
```
real, pointer :: array(:)
real, pointer :: array2(:)

allocate(array(6))
```

Overlapping memory



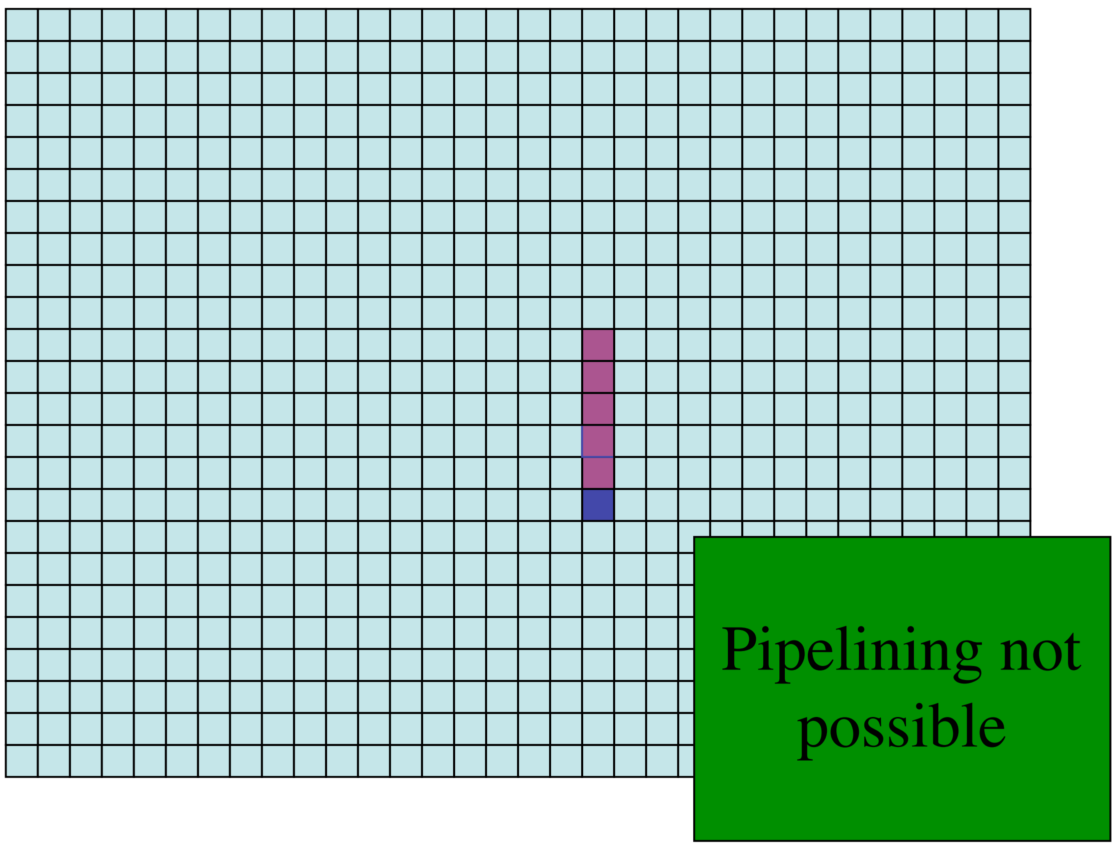
```
real, pointer :: array(:)
real, pointer :: array2(:)

allocate(array(6))
array2=>array(2:6)
```

Overlapping memory



```
real, pointer :: array(:)
real, pointer :: array2(:)
allocate(array(6))
array=0;
array2=>array(2:6)
Array(1)=2.
do I=2,5
  array2(I)=array2(I-1)+
    array(I-1)
end do
```

Overlapping memory




```
real, pointer :: array(:)
real, pointer :: array2(:)
allocate(array(6))
array=0;
array2=>array(2:6)
Array(1)=2.
do I=2,5
  array2(I)=array2(I-1)+
    array(I-1)
end do
```

Complex numbers

```
complex :: a(:),b(:),c(:)
do i=1,n
  c(i)=a*b
end do
```

a is real

```
complex :: a(:),b(:),c(:)
do i=1,n
  c(i)=cmplx(real(a(i))*real(b(i)),&
    real(a(i))*aimag(b(i)))
   end do
```

1/3 the number of operations

Other potential pitfalls

- Print statements
- Passing pointers
- I/O

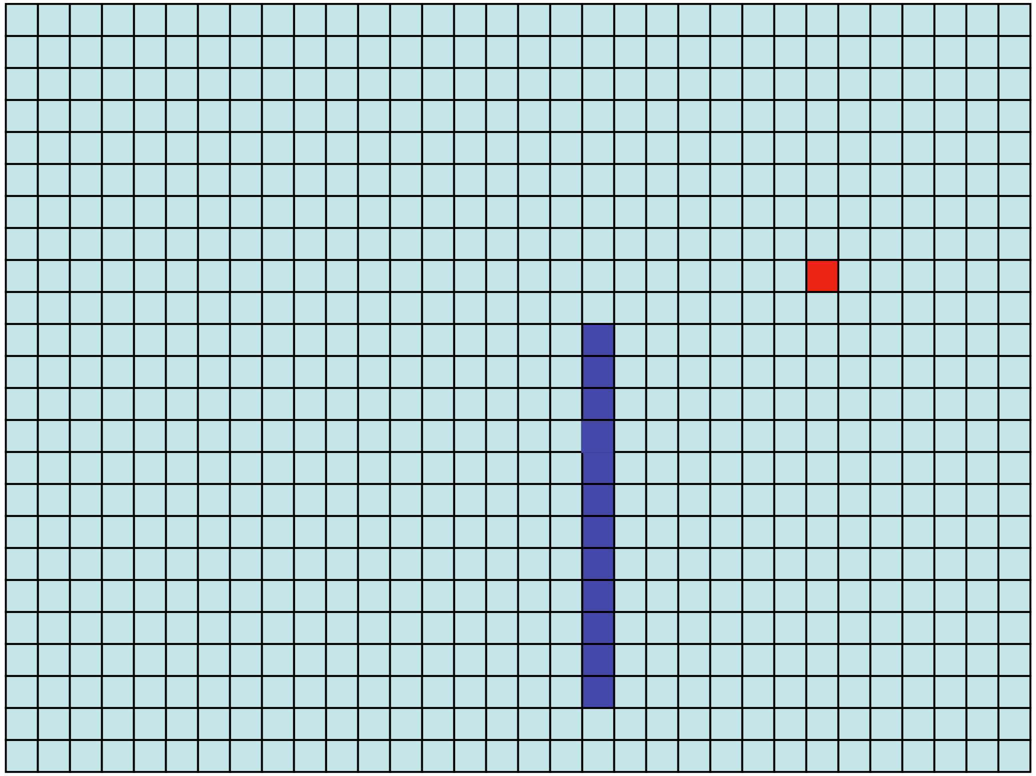
Print statements

- Even the possibility (in an if statement) can have a dramatic effect on the speed of a code
- Stops many optimizations because many optimizations are not possible
- Often forces a “flush” which will have a negative effect on cache performance

C (and C++)

- When you have structures, pass them as a pointer
  - Speed difference can be a factor of 50 in some cases
  - A killer in an inner loop

Passing structures in C



```
struct alpha
   int a;
   float b[100][100];
   float c[100][100];
} _alpha;
```

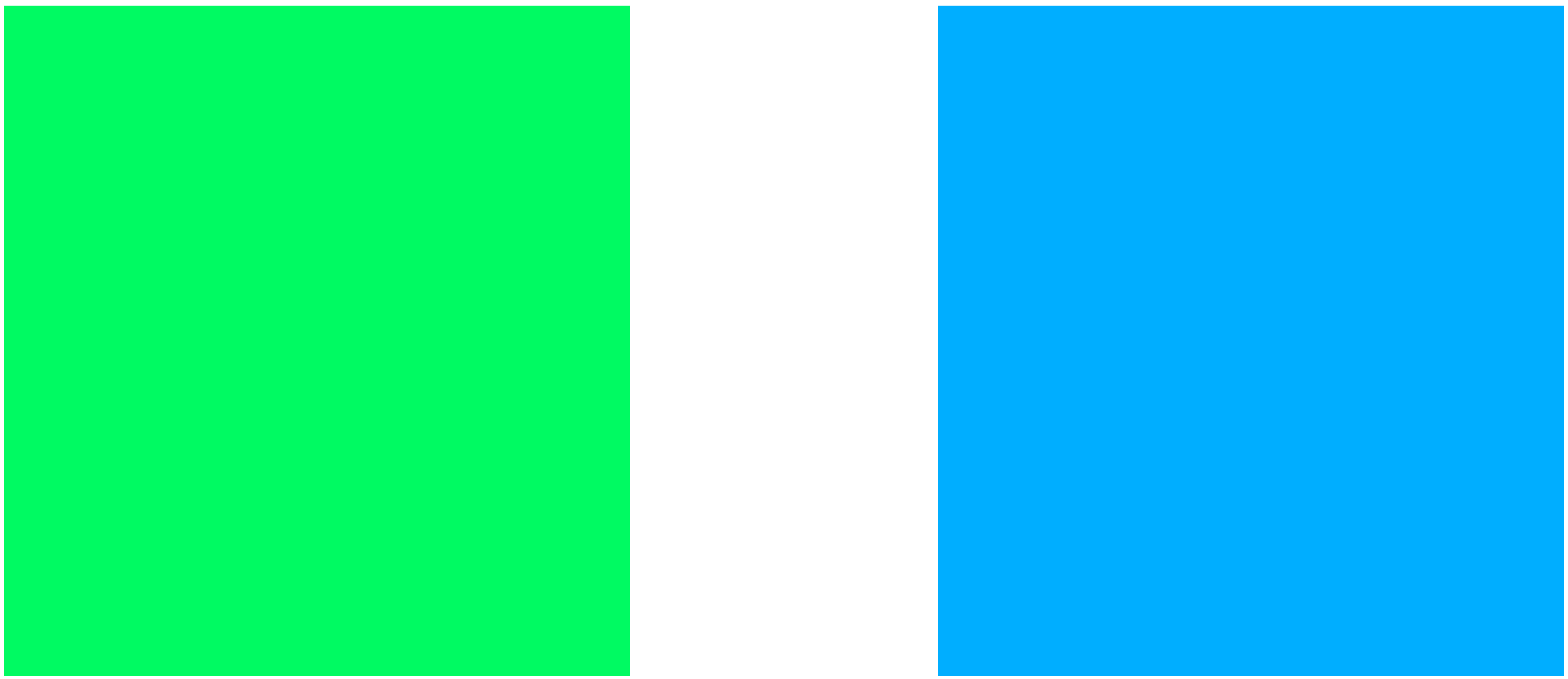
What you are copying if you pass a structure

What you are copying if you pass a pointer

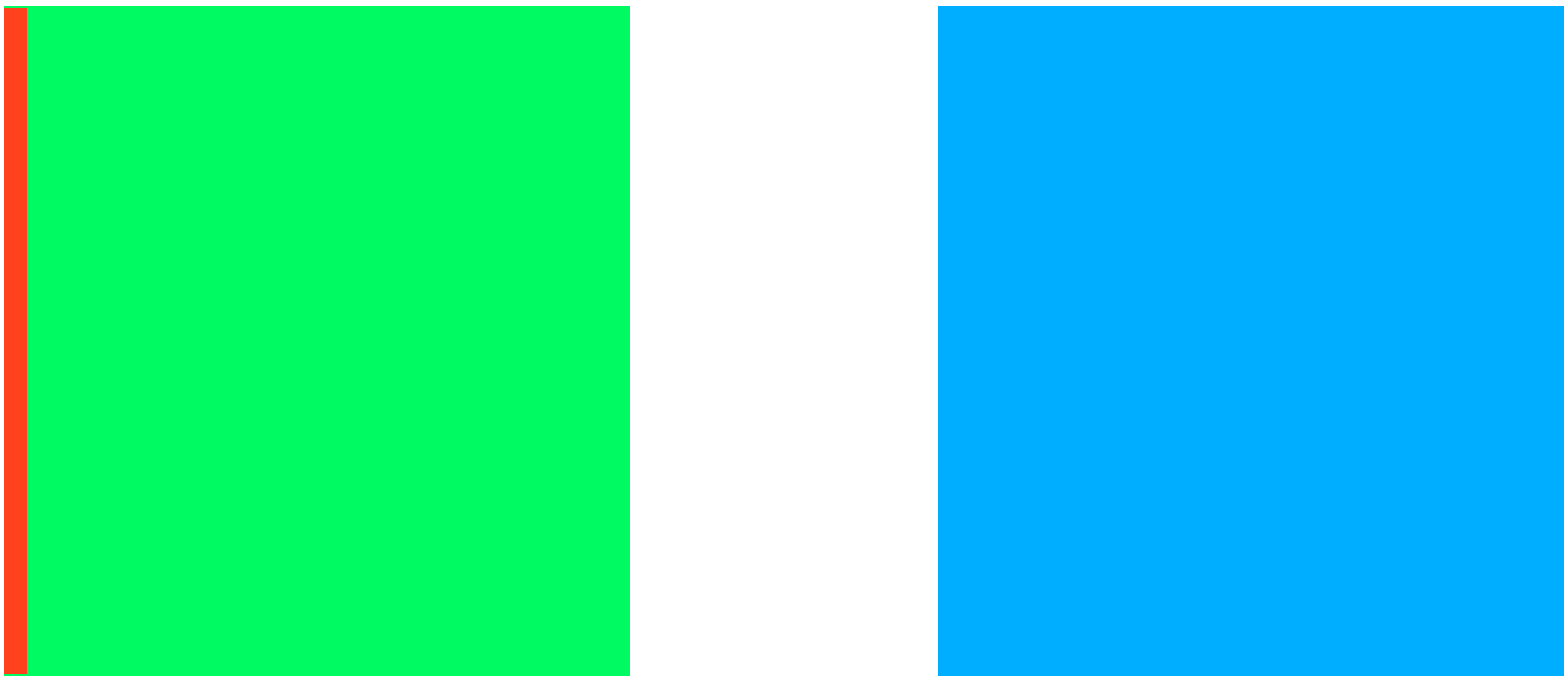
I/O

- For maximum speed try to follow the same rules that are effective in maximizing clock cycles
  - Large rather than small reads (the cost of initializing a read is non-trivial)
  - Avoid seeking within a file (a seek is faster than a read but continuous seeking will kill performance)
  - Do binary read/write rather than ASCII whenever possible (matlab factor 100+)

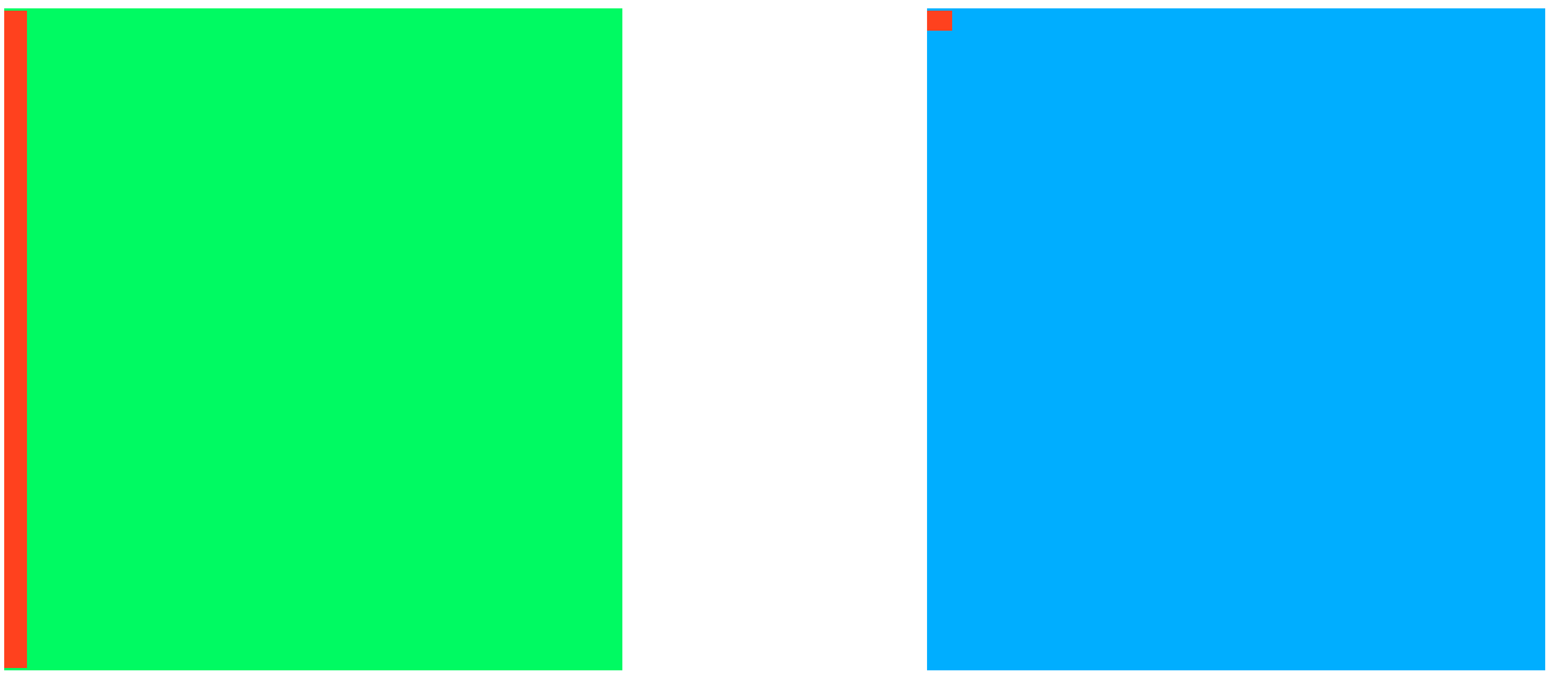
Example: Out of core transpose



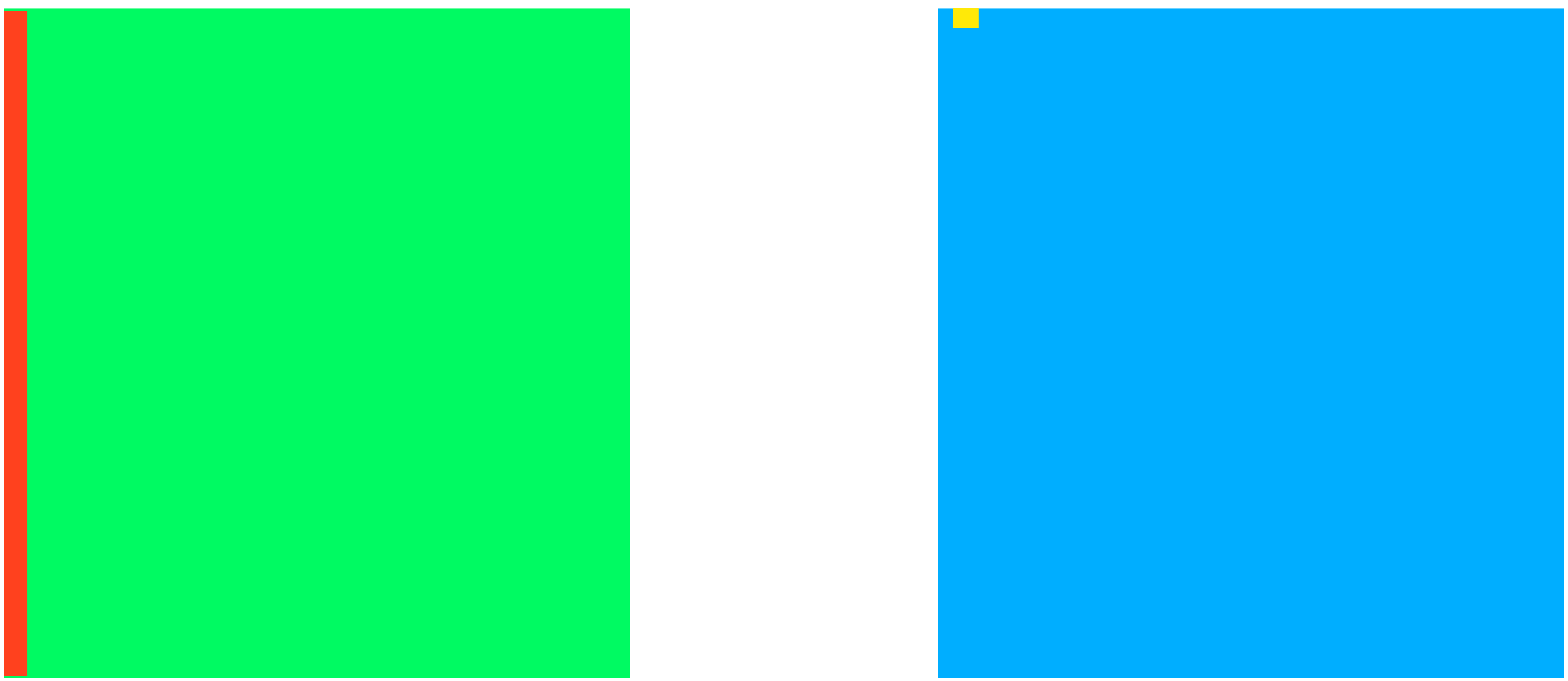
Read



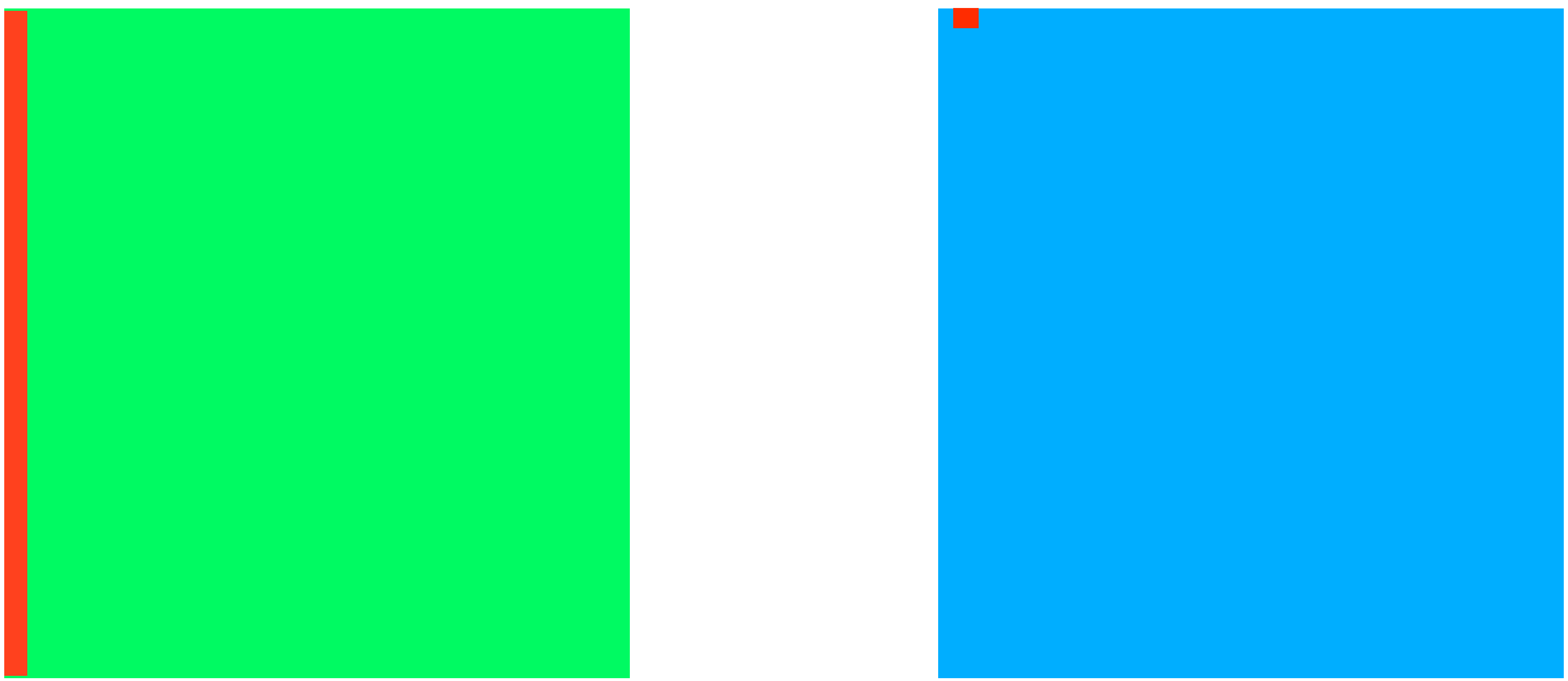
Write



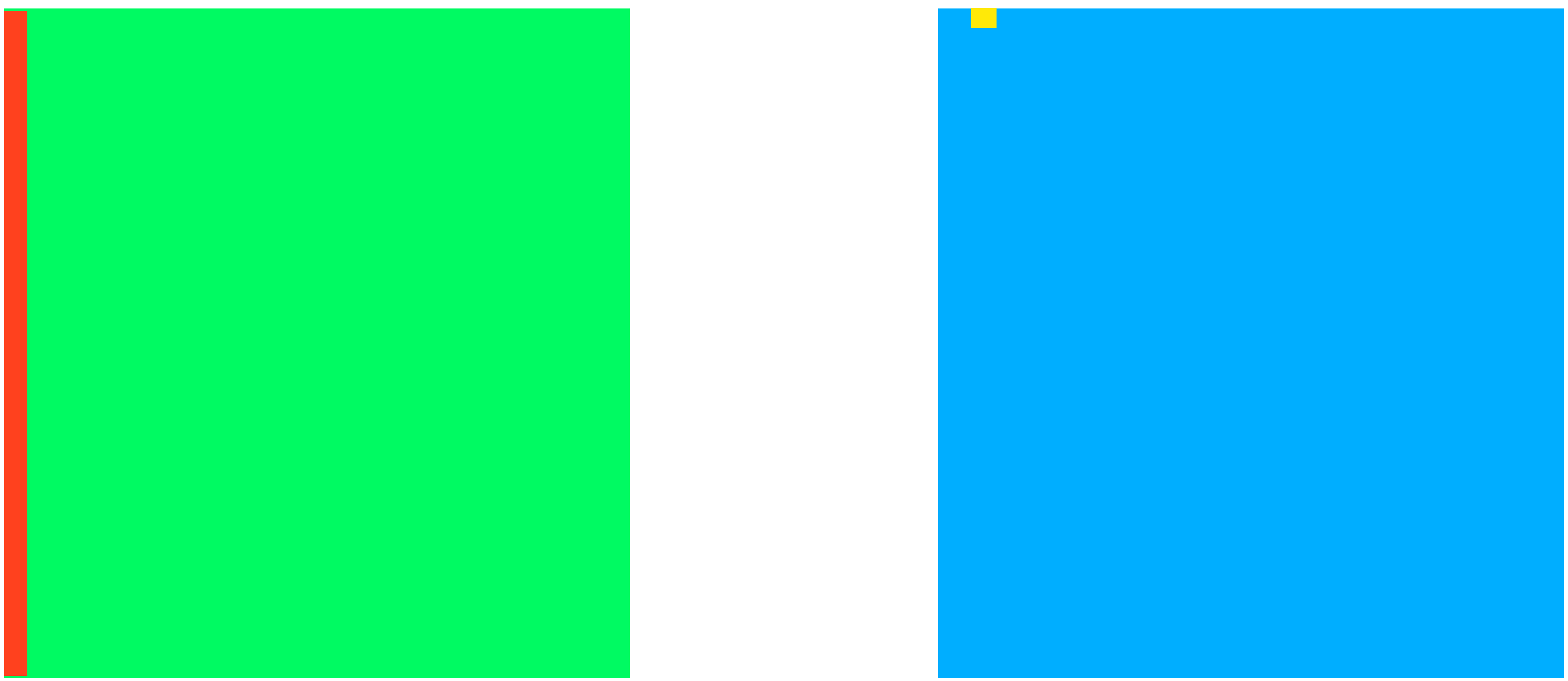
Seek



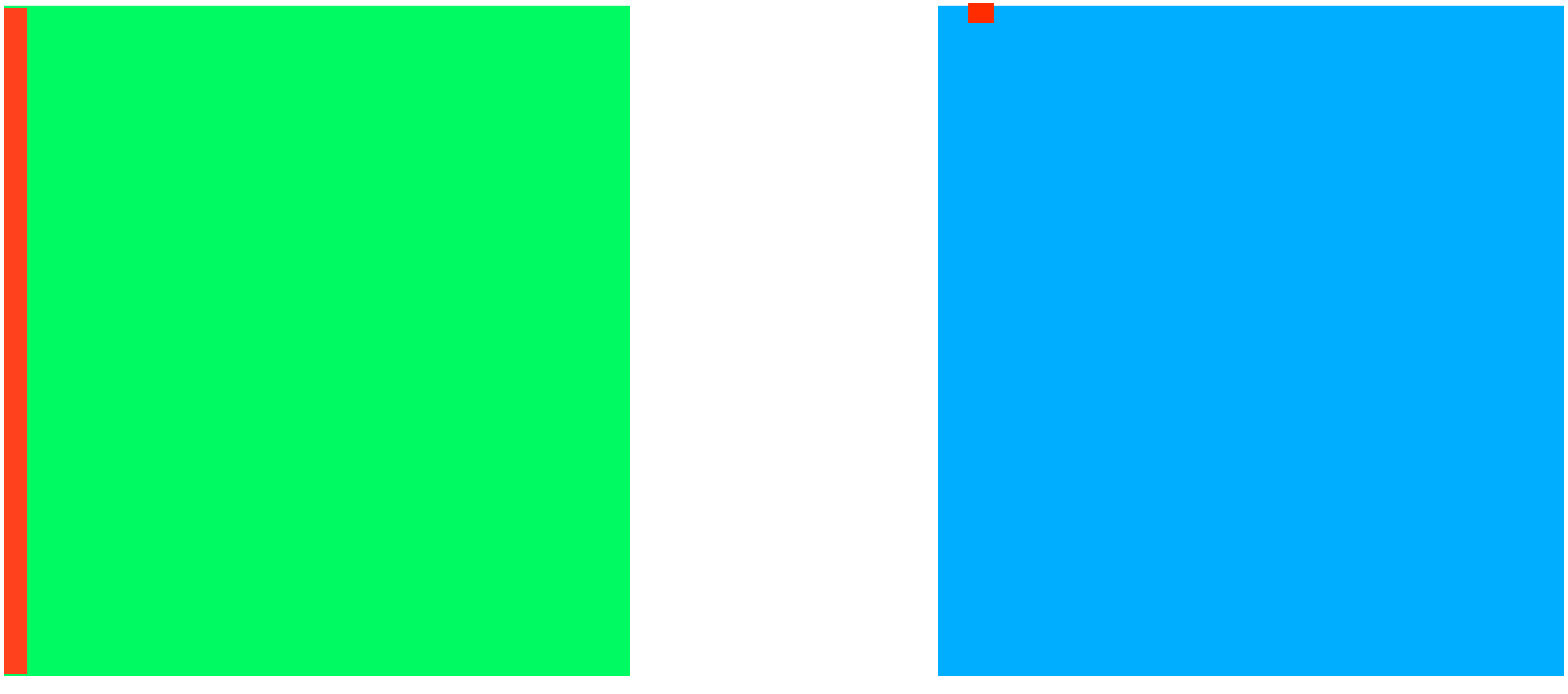
Write



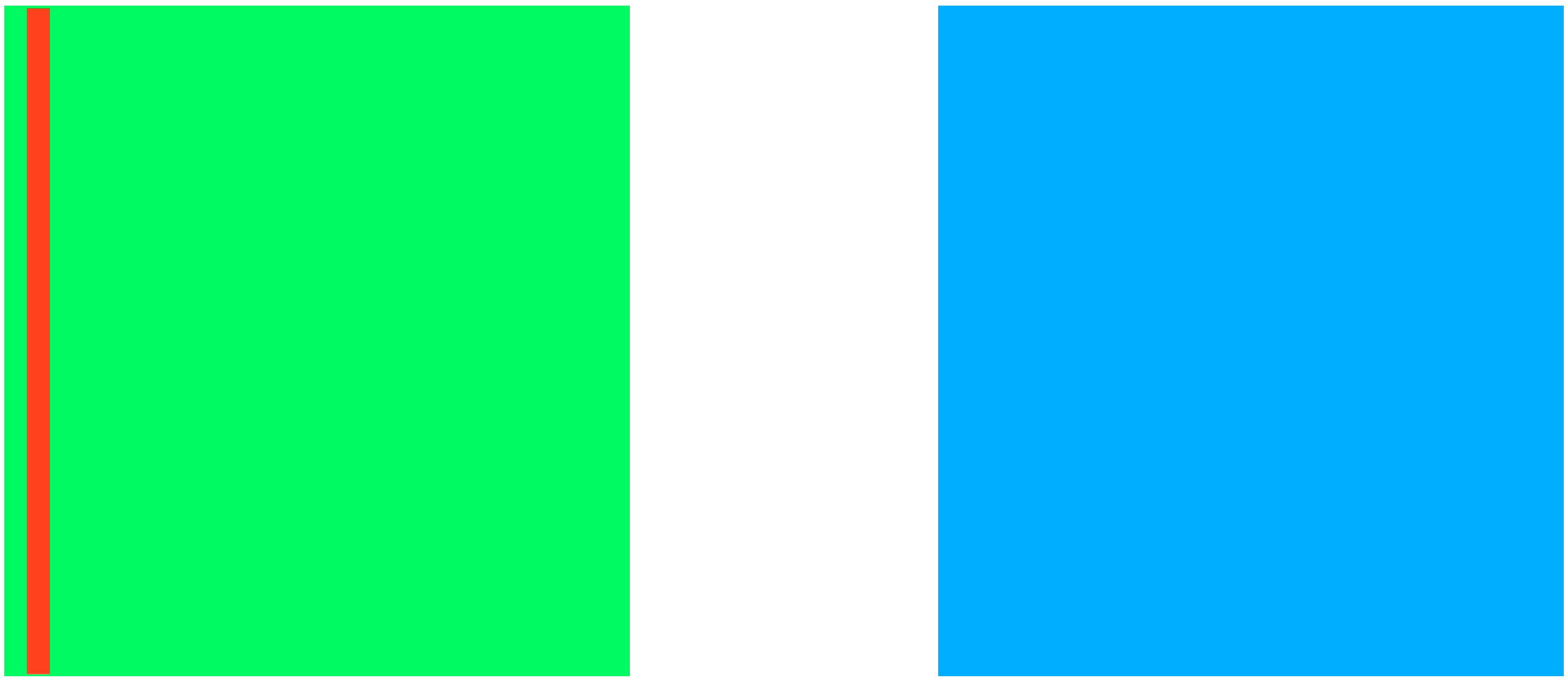
Seek



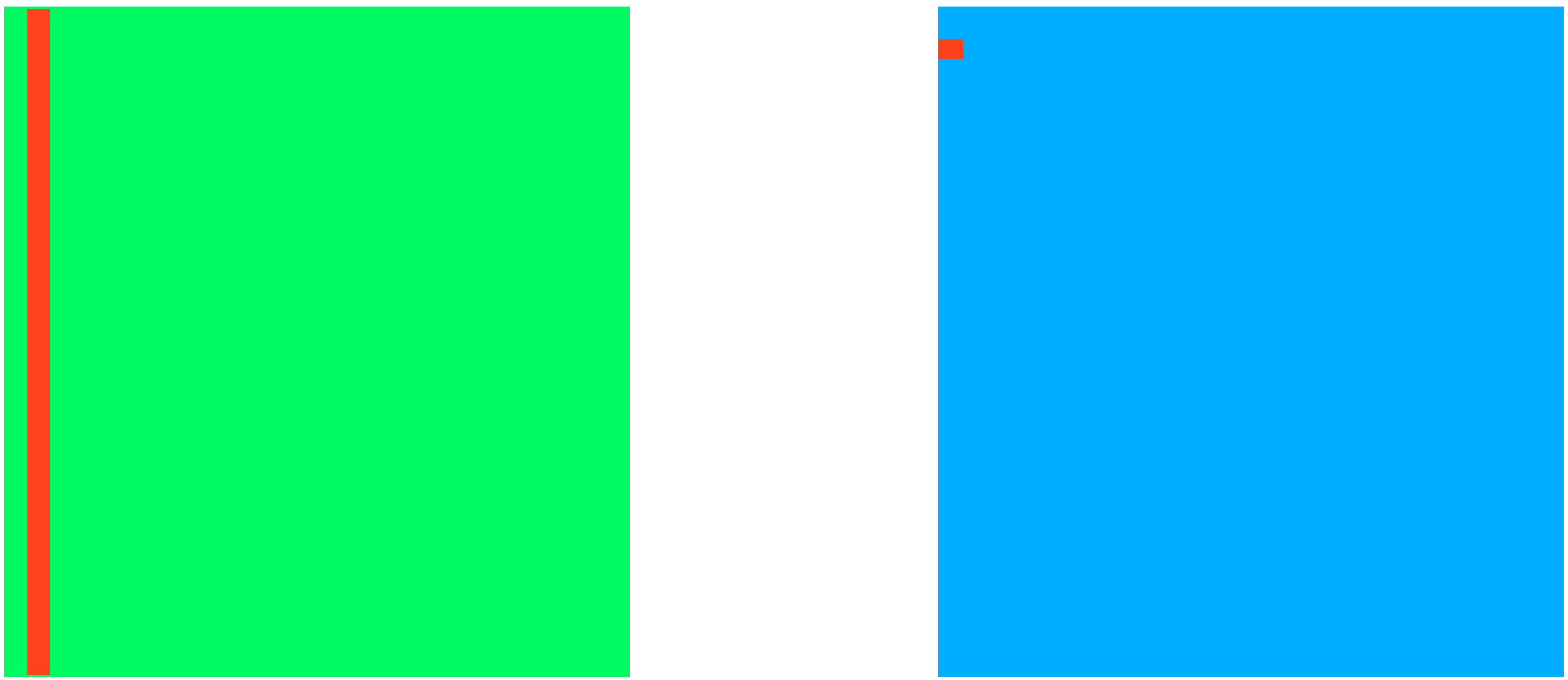
Write



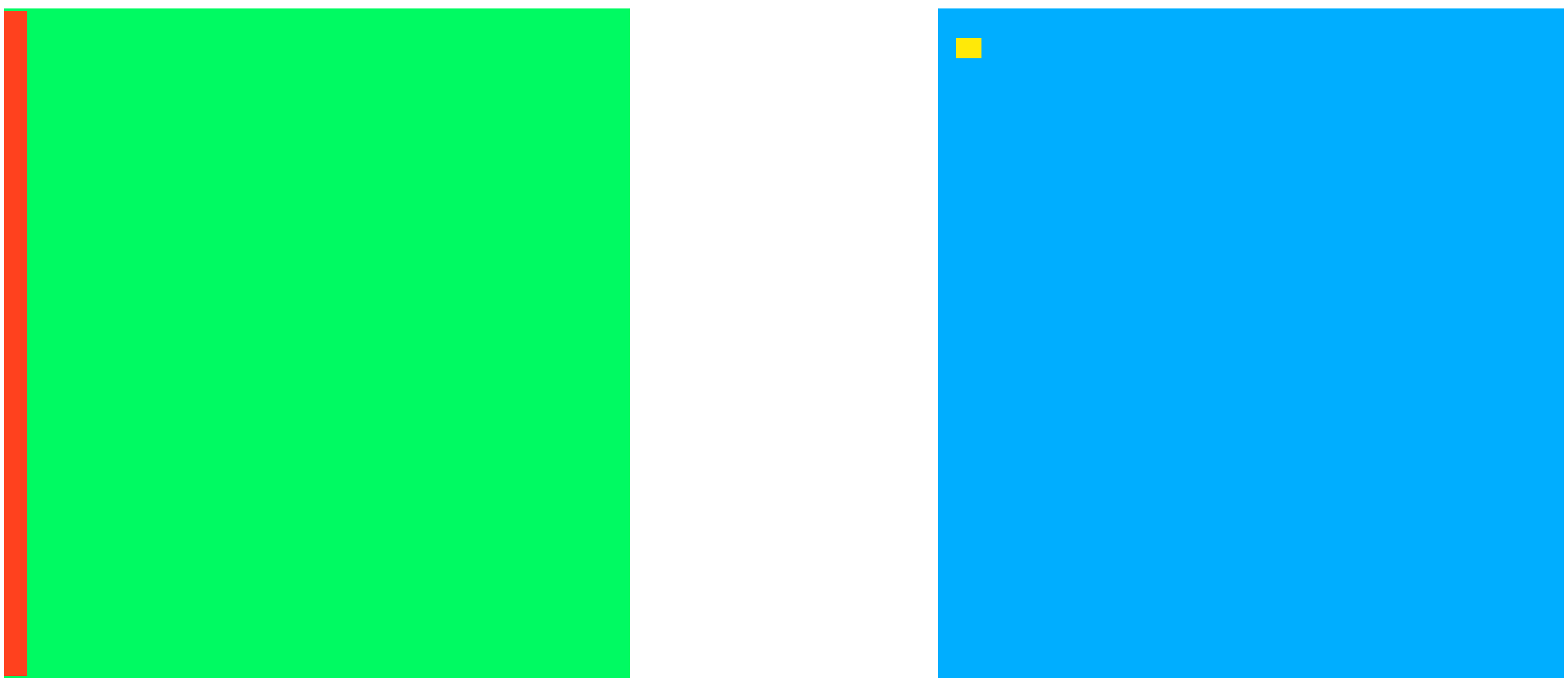
Read



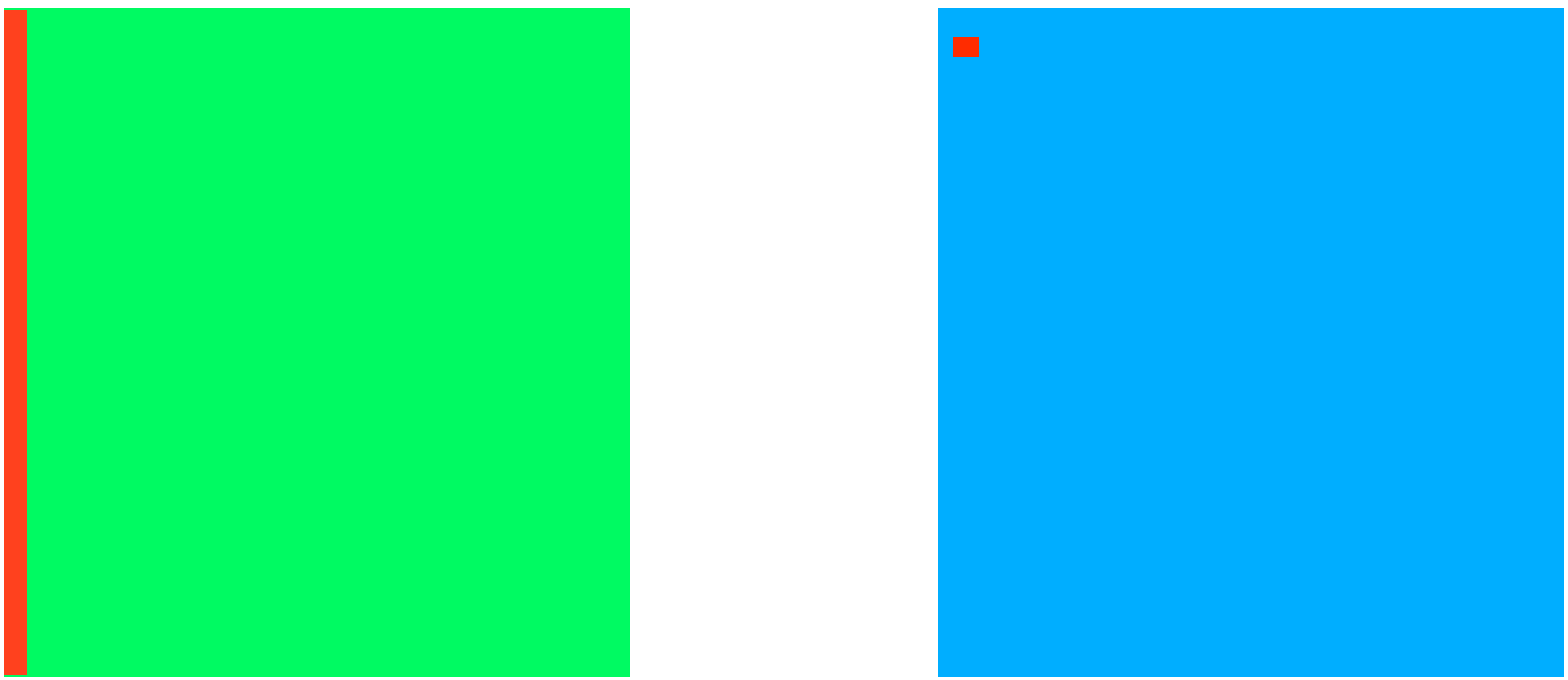
Write



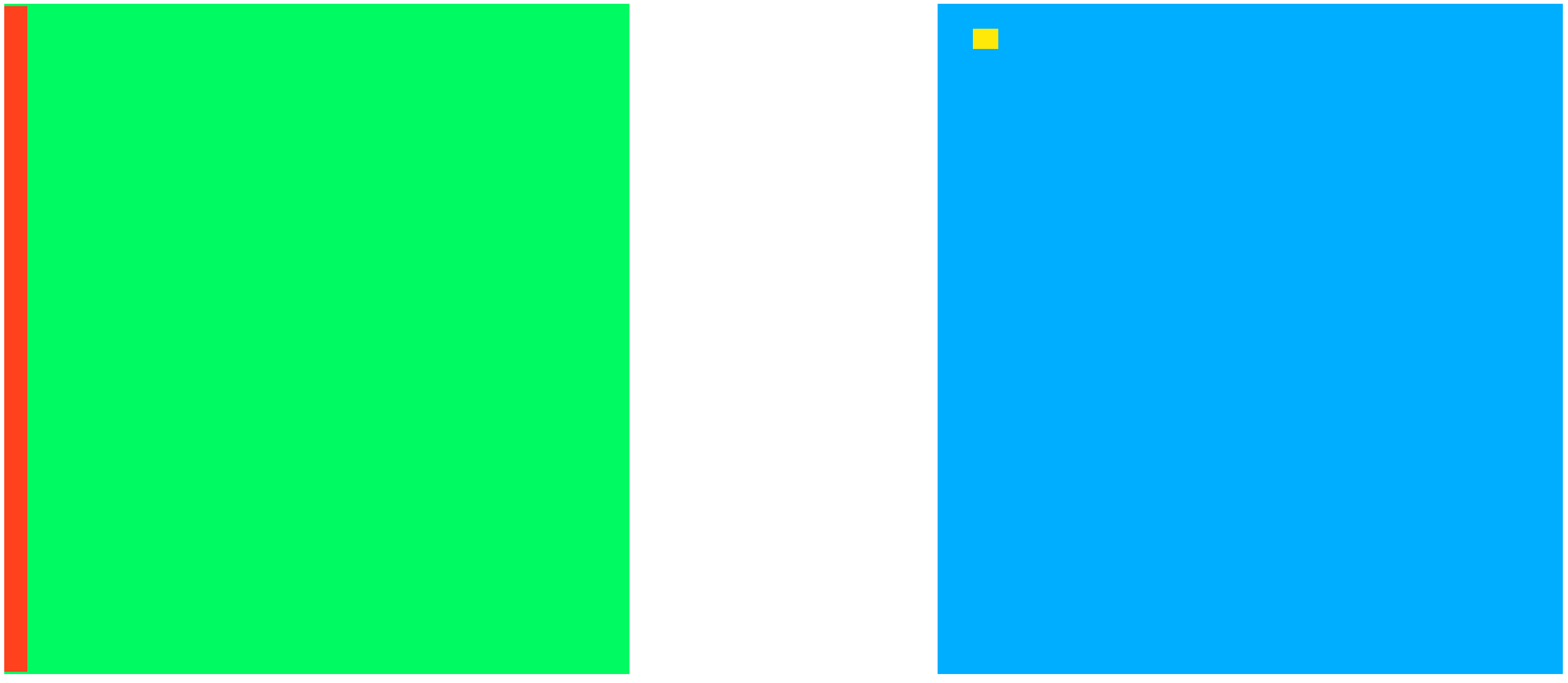
Seek



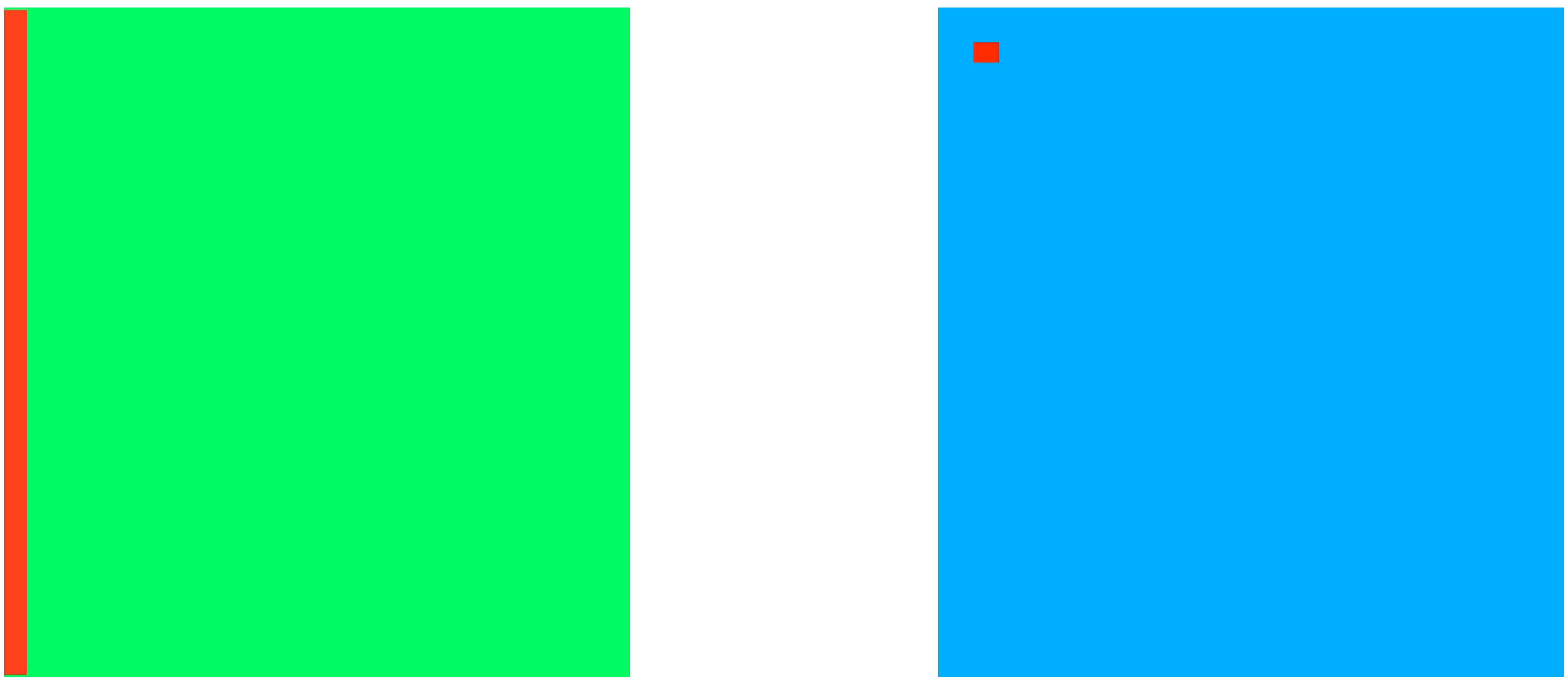
Write



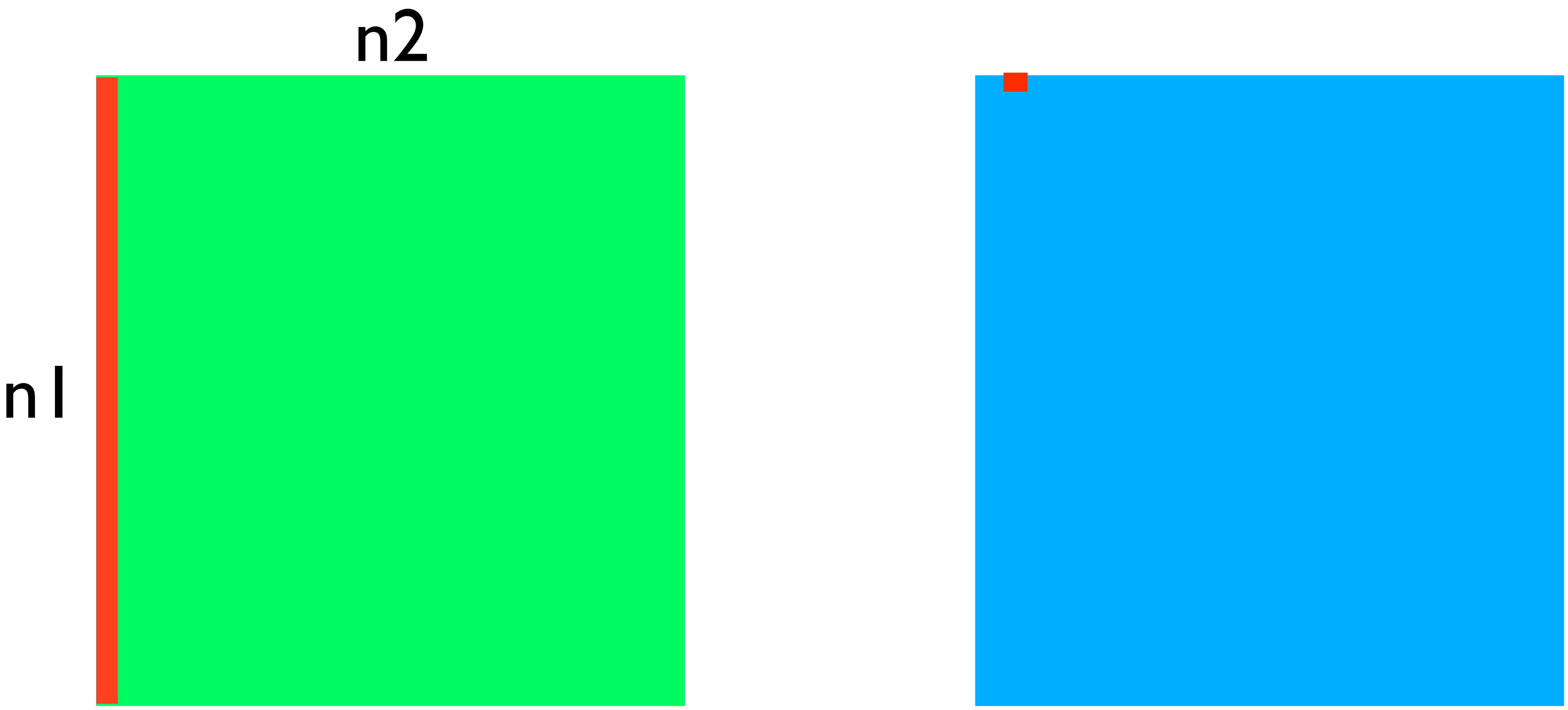
Seek



Write

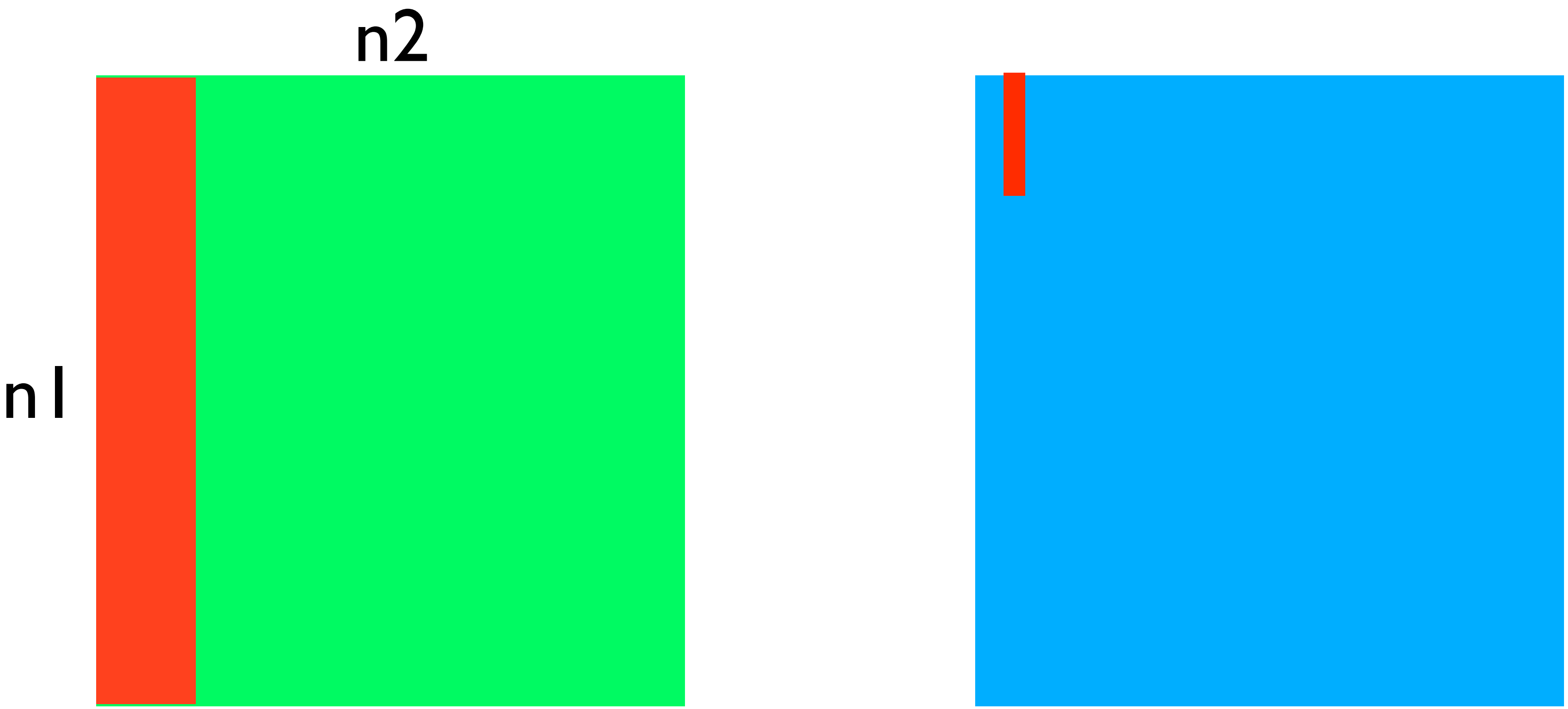


Cost



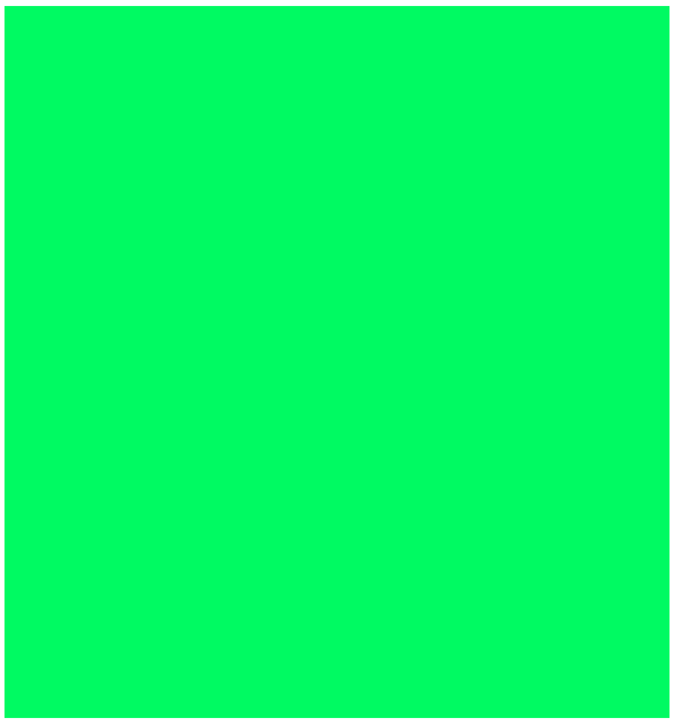
n2 reads of size n1
n1*n2 seeks of size 1
n1*n2 writes of size 1

Cost: 5x read size

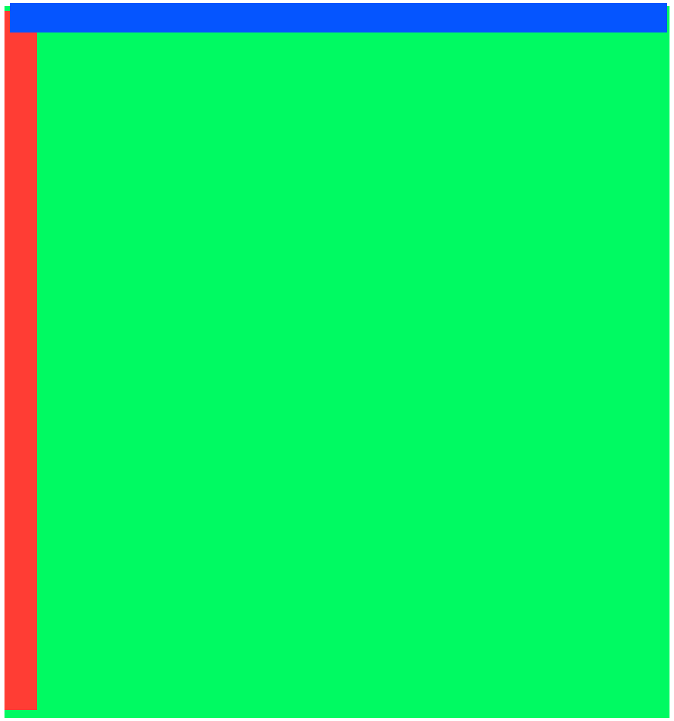


n2/5 reads of size n1*5
n1*n2/5 seeks of size 5
n1*n2/5 writes of size 5

Sparse matrix-vector multiplication

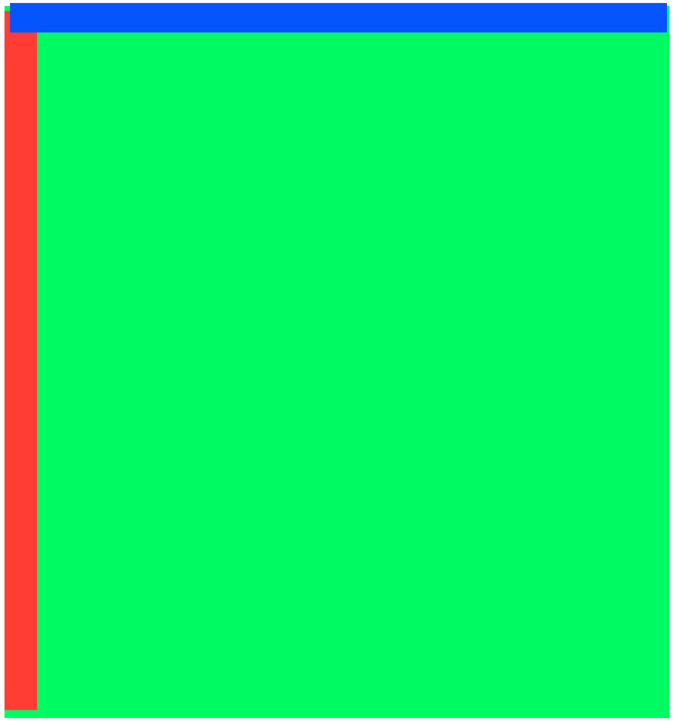


Sparse matrix-vector multiplication



Input vector
Output vector

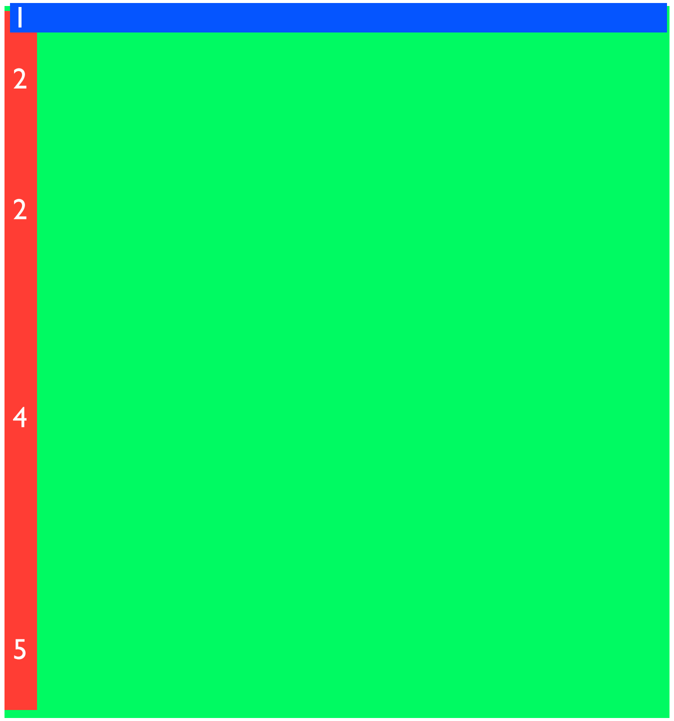
Sort non-zero elements



Input vector
Output vector

Minimize cache misses by sorting non-zero elements.

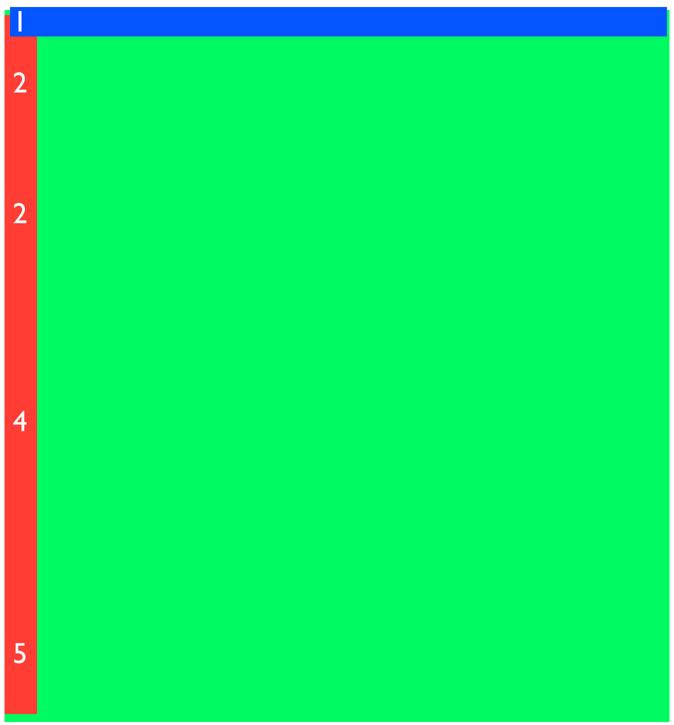
Minimizes in misses in input or output



Input vector
Output vector

Minimize cache misses by sorting non-zero elements.

Minimizes in misses in input or output

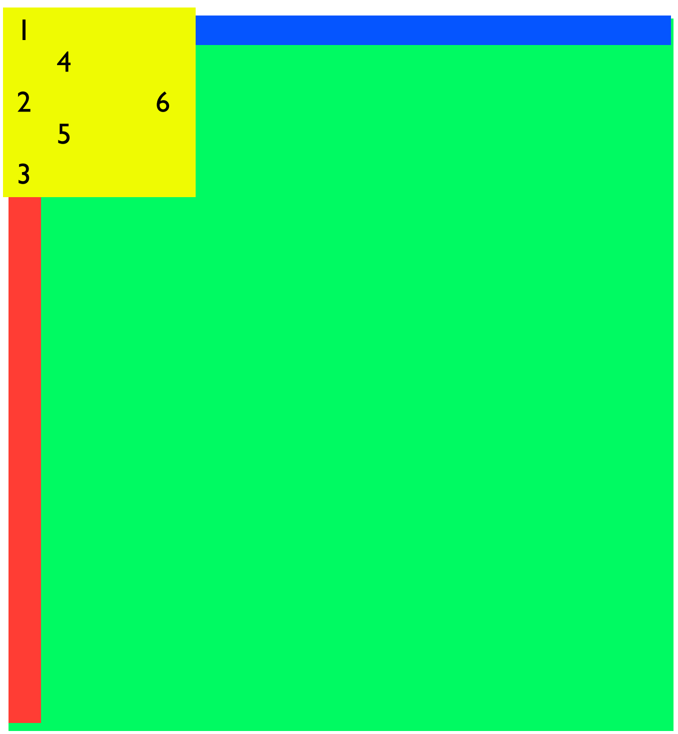


Input vector

Output vector

Consistent cache misses in the output vector

Blocking part 1



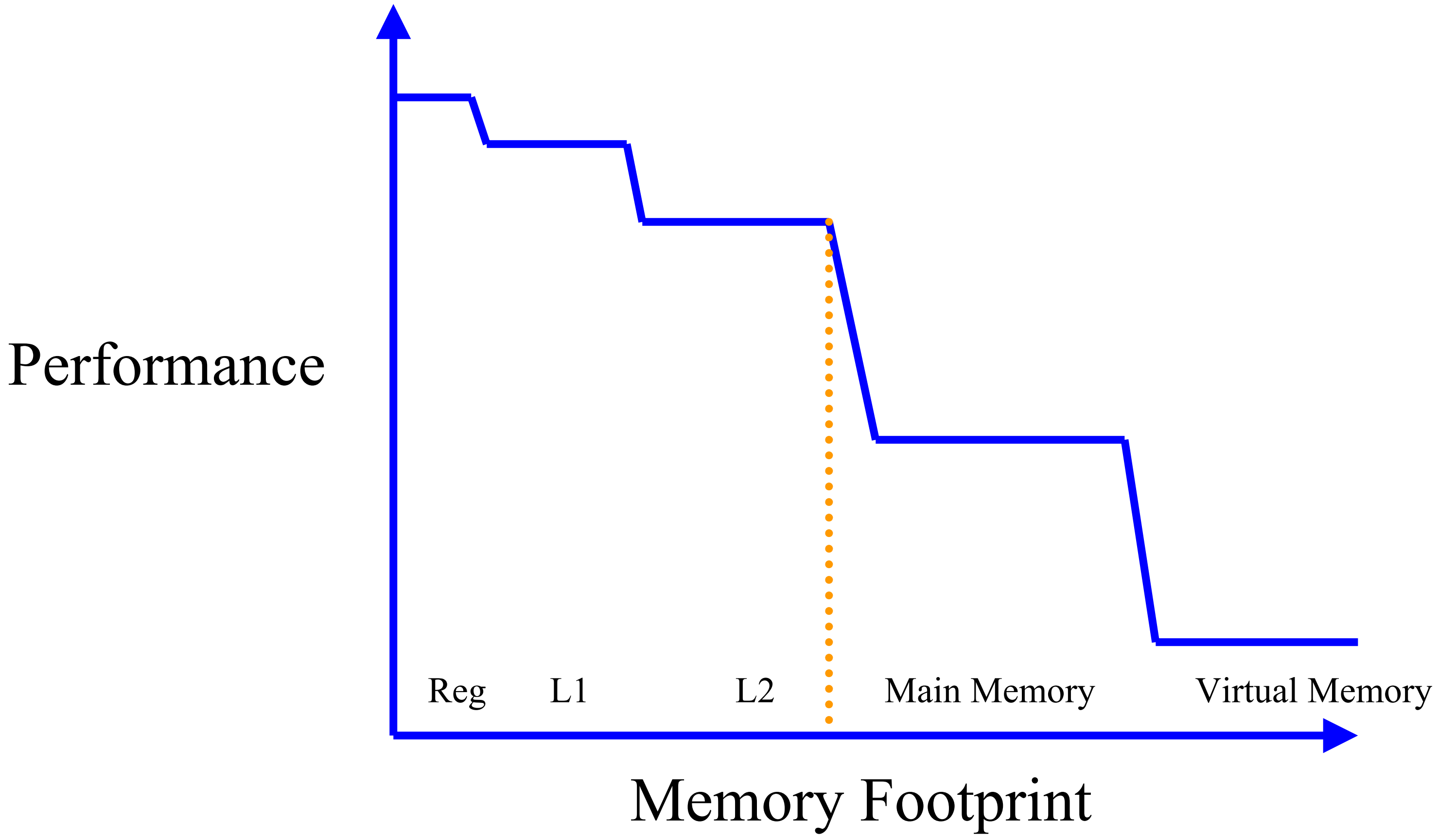
Input vector

Output vector

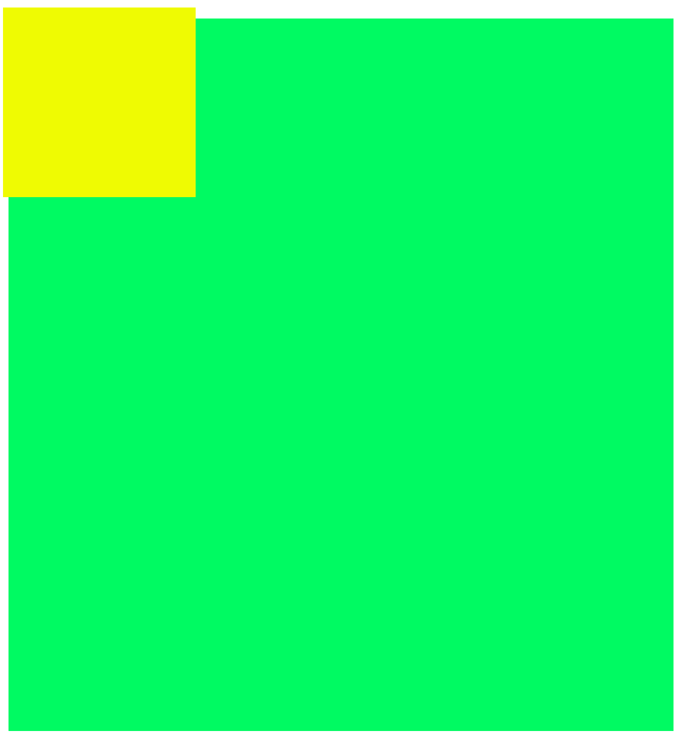
Minimizes misses in both input and output

Performance vs. Memory footprint

Fastest performance will be achieved when most of the data is in the cache.



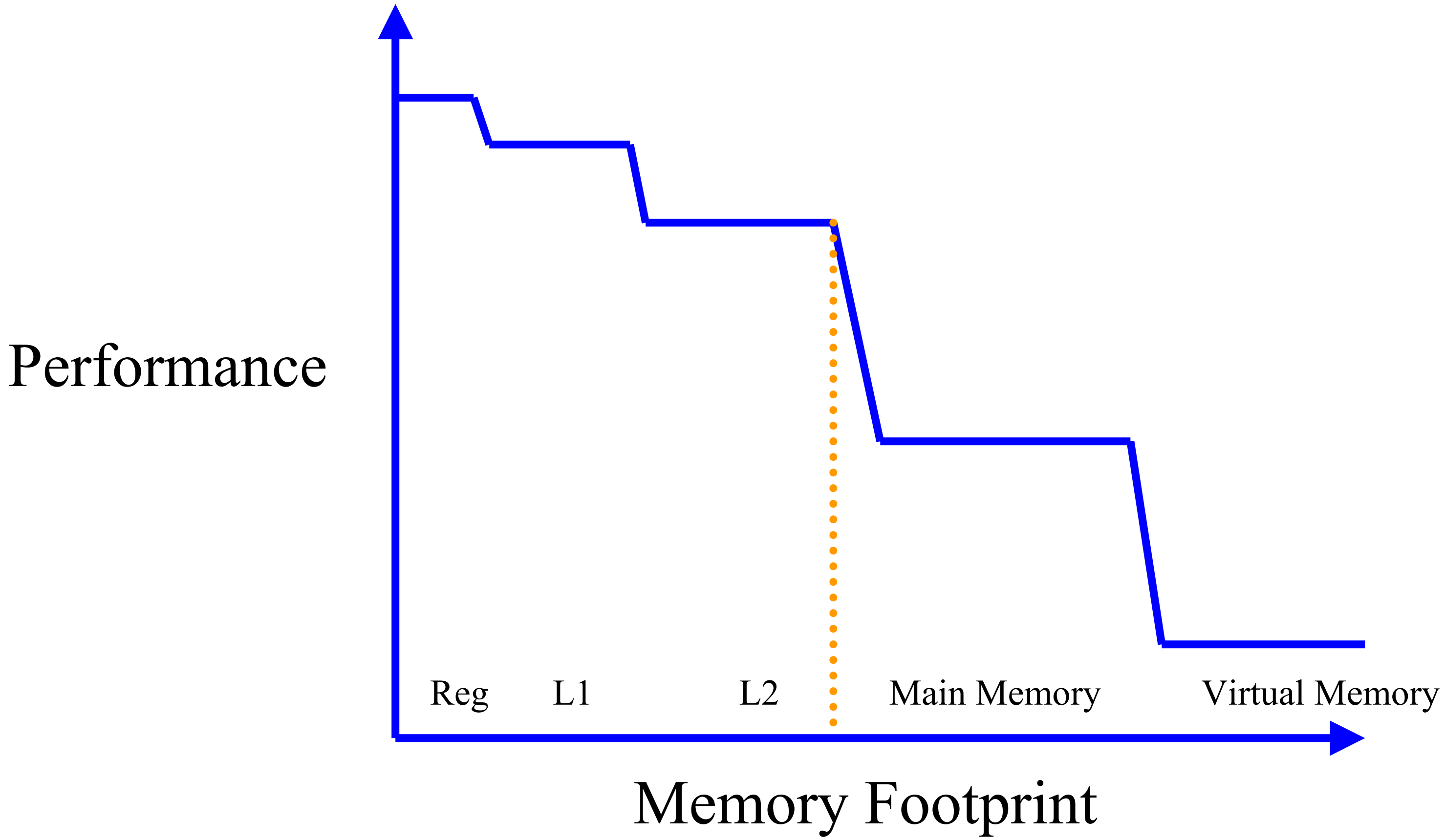
Convolution: Blocking part 2



Break into blocks that all fit into cache

Performance vs. Memory footprint

Fastest performance will be achieved when most of the data is in the cache.



Vector operations

- Write everything so pipelining is possible
- No automatic arrays
- Use machine libraries

Analyzing performance

- Testing on a small version of your problem is often not a valid test
- Some analyzing tools distort the possible optimizations and give inaccurate results
- Often the most reliable means is through timers

Basic rules for efficient coding

- Avoid conditionals in loops
- Avoid functions in loops
- Precompute as much as possible
- Try to stay in cache as much as possible
- Do table lookups on very expensive operations
- Make the code as simple as possible so the compiler will do the work for you

Useful external libraries

- Blas
- Lapack
- Arpack
- FFT

Basic Linear Algebra Subprograms

- Scalar and vector operations (level 1)
  - $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$
- Matrix-vector operators (level2)
  - $\mathbf{y} = \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$
- Matrix-matrix operators (level 3)
  - $\mathbf{C} = \alpha\mathbf{A}\mathbf{B}\mathbf{x} + \beta\mathbf{C}$

Linpack

- Aimed at solving sets of linear equations with full matrices
- Relies on Blas routines
- Provides an out-of-the-box tool to begin doing inversion
- Limited by the problem it is addressing
  - Dense matrix that can be hold in memory

Arnoldi package (ARPACK)

- Designed to solve large scale eigenvalue problems
- Efficient for sparse or structured matrices
- Not efficient for dense matrices

BLAS

- Used as building block for Linpack
- Level 1 and Level 2 are not that efficient on current hardware (they were designed more for an array processor)
  - Compiler can often achieve a factor 2+ improvement
- Level 3 is fairly efficient especially on a SMP machine

FFT

- FFTW
  - Efficient package for computing FFTs on a variety platforms
- Sun performance library/Intel Math Kernel library
  - Are more efficient, but not as portable code
  - Significant performance improvements on power of 2 arrays