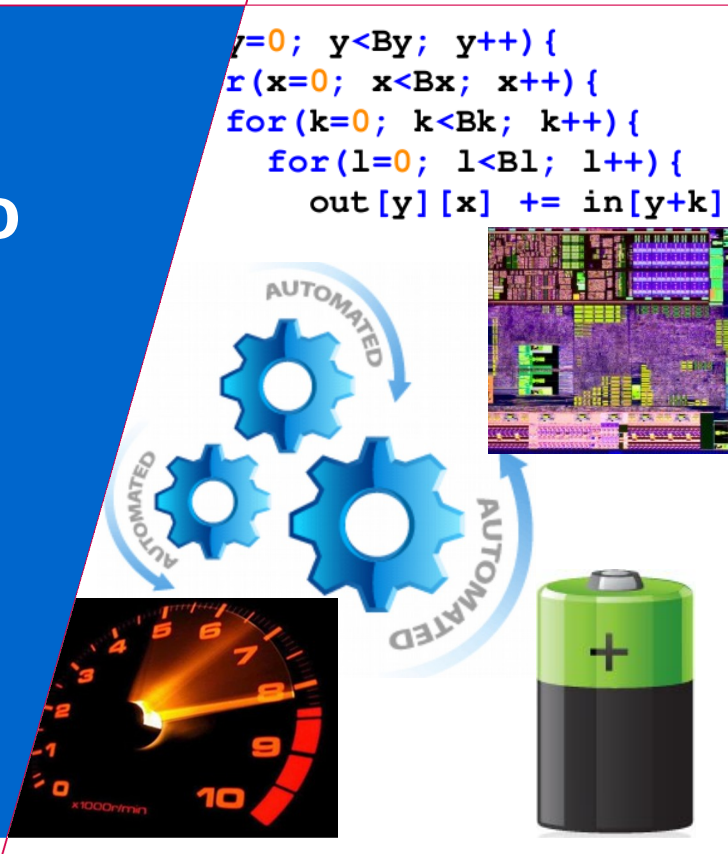
Electronic Systems

Tricks of the Trade: From Loop Transformations to Automatic Optimization



By: Maurice Peemen

5KK73 Embedded Computer Architecture

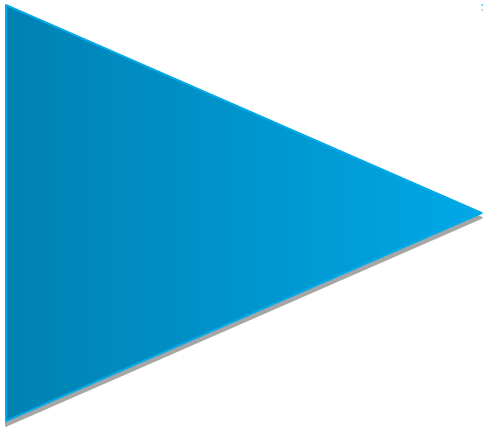
TU/e Technische Universiteit Eindhoven University of Technology

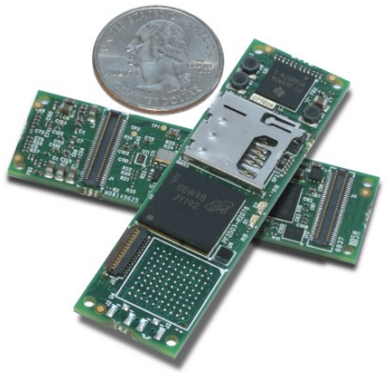
Where innovation starts

Application Excecution Time

- High performance computing
- Embedded computing
- Small portions of the code dominate execution time
- Hot spots
- Loops
- Improve the small part
- Loop Transformations

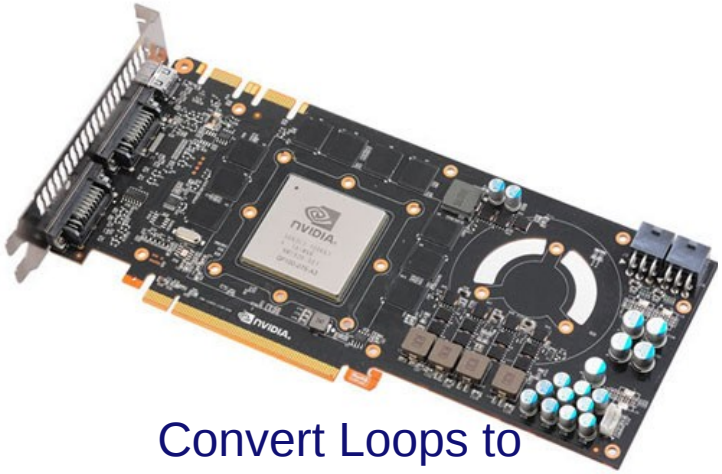
```
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    for(k=0; k<N; k++)
      c[i][j] += a[i][k] * b[k][j];
}}
```





Loop Transformations

- **Can help you in many compute domains**



Convert Loops to
GPU Threads

Embedded Devices
Fine-tune for architecture

CPU Code
Parallelism
Memory Hierarchy

HLS languages for HW

Loop Transformation Classes

- **Data-Flow-Based Transformations**
- **Iteration Reordering Transformations**
- **Loop Restructuring Transformations**

Data-Flow-Based Transformations

- Can be very useful:
  - Simple compiler
  - Simple architecture
    - No Hardware Multiplier
    - No Out-Of-Order Superscalar
    - No Speculative Branch Prediction
  - If you are in a hurry and missing a few percent
- These are the easy tricks
- To get ready for the more difficult ones

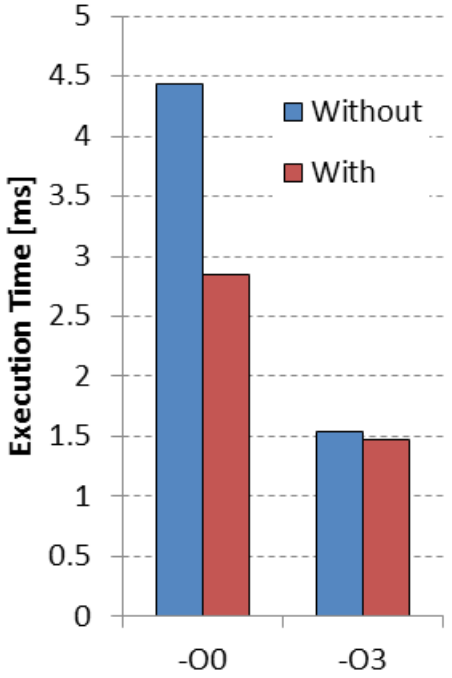
*** Examples are from:**
D.F. Bacon, S.L. Graham, and O.J. Sharp, *Compiler transformations for high-performance computing*. ACM Comput. Surv. 26, (1994)

Loop-Based Strength Reduction

- Replace an expensive operation
- Can be multiplication, especially for embedded cores
- No HW multiplier, e.g. MicroBlaze
- An optimizing compiler can help

```
c=3;
for(i=0; i<N; i++){
  a[i] = a[i] + i*c;
}
```

```
c=3;
T=0;
for(i=0; i<N; i++){
  a[i] = a[i] + T;
  T = T + c;
}
```

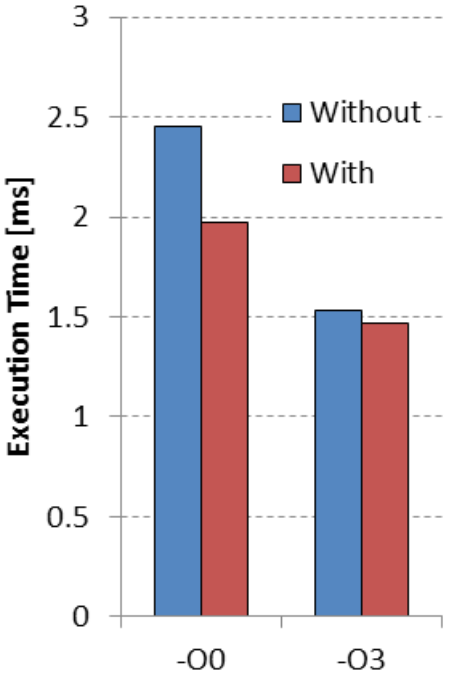


Induction Variable Elimination

- Induction variable?
  - Compute loop exit conditions
  - Compute memory addresses
- Given a known final value and relation to addresses
  - Possible to eliminate index calculations
- Again an optimizing compiler can help

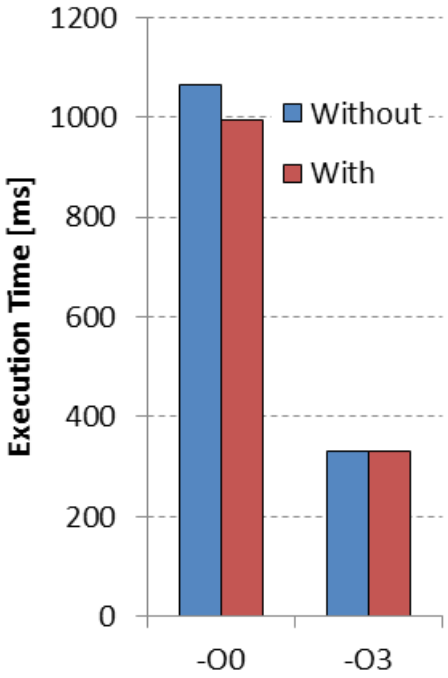
```
for(i=0; i<N; i++){
  a[i] = a[i] + c;
}
```

```
A = &a[0];
T = &a[0] + N;
while(A<T){
  *A = *A + c;
  A++;
}
```



Loop-Invariant Code Motion

- Index of c[] does not change
- Move one loop up and store in register
- Reuse it over iterations of inner loop
- Most compilers can perform this
- Example is similar to
  Scalar Replacement Transformation

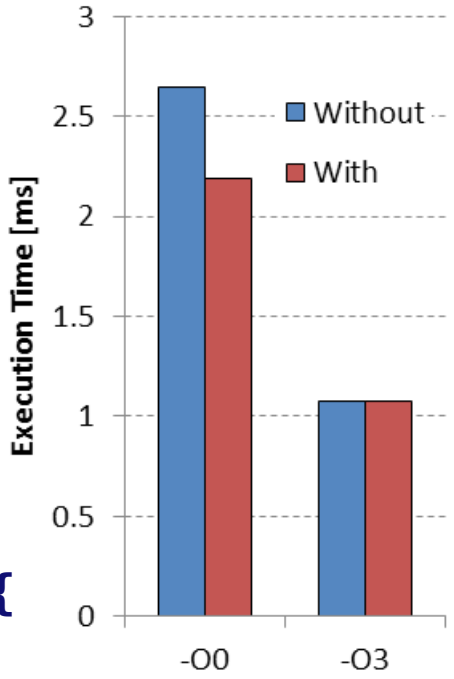



```
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    a[i][j]=b[j][i]+c[i];
  }
}
```

```
for(i=0; i<N; i++){
  T=c[i];
  for(j=0; j<N; j++){
    a[i][j]=b[j][i]+T;
  }
}
```

Loop Unswitching

- Conditional execution in body
- Speculative branch prediction helps but
- This branch is loop invariant code
  - Move it outside the loop
  - Each execution path independent loops
- Increases code size

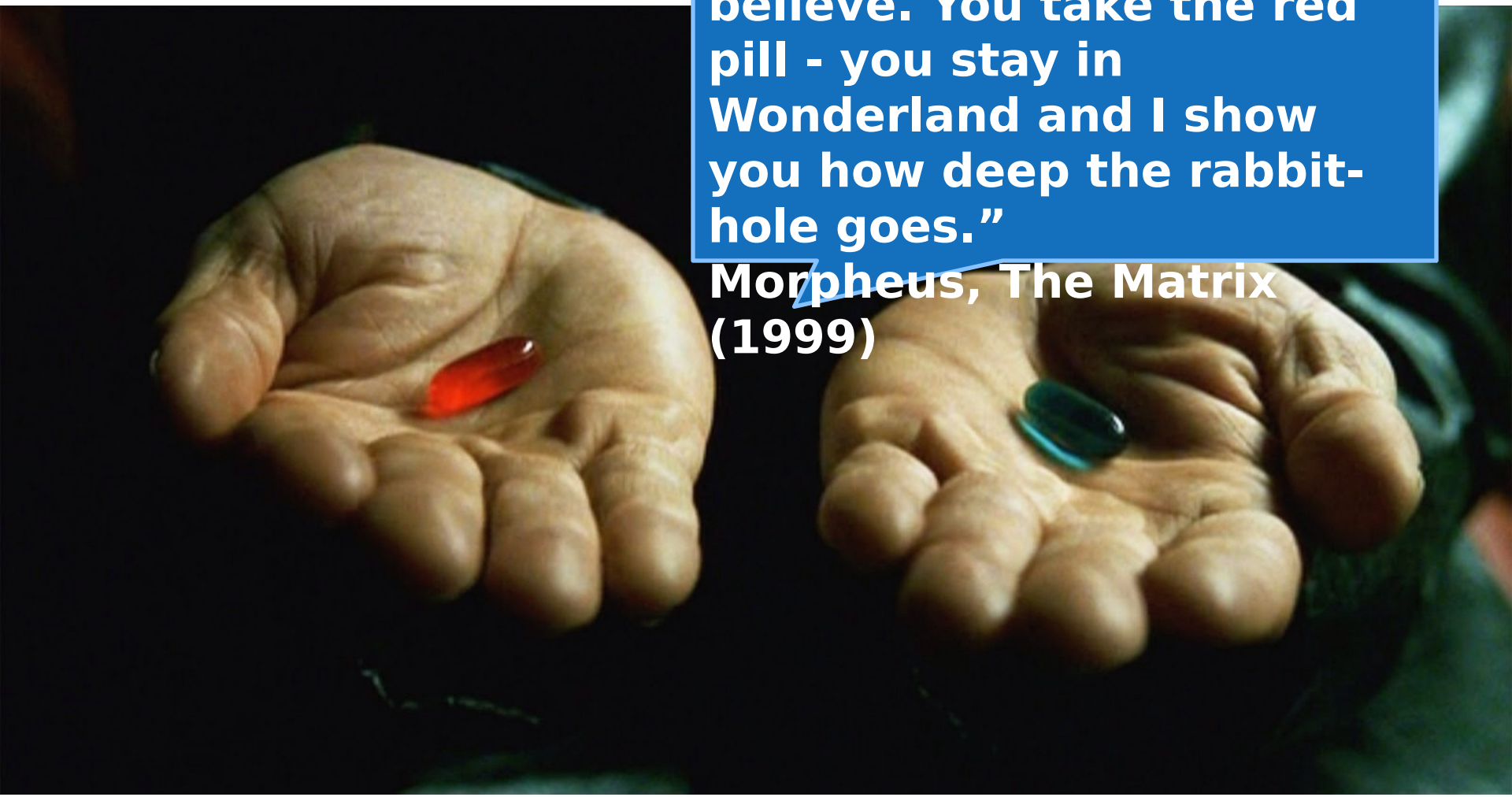



```
for(i=0; i<N; i++){
  a[i] = b[i];
  if(x<7){a[i]+=5;}
}
```

```
if(x<7){
  for(i=0; i<N; i++){
    a[i]=b[i];
    a[i]+=5;
  }
}
else{
  for(i=0; i<N; i++){
    a[i]=b[i];
  }
}
```

This is only the start

“You take the blue pill - the story ends, you wake up in your bed and believe whatever you want to believe. You take the red pill - you stay in Wonderland and I show you how deep the rabbit-hole goes.”
Morpheus, The Matrix (1999)



Loop Transformation Classes

- Data-Flow-Based Transformations
- Iteration Reordering Transformations
- Loop Restructuring Transformations

Iteration Reordering Transformations

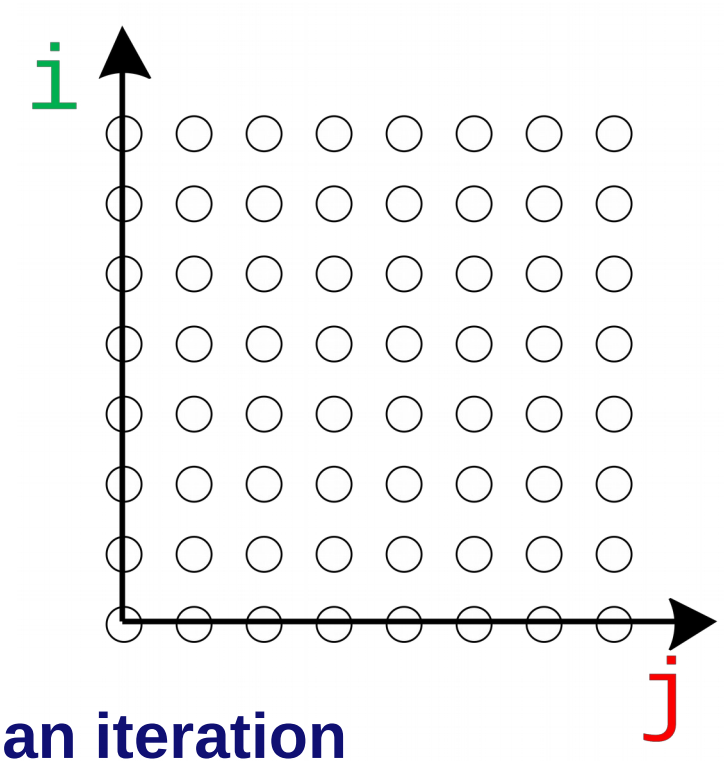
- Change the relative ordering
  - Expose parallelism by altering dependencies
  - Improve locality by matching with a memory hierarchy
- We need more theory to understand these
- Or combine the tricks to form a methodology
- Normal compilers don't help you with this
- Only state-of-the-art compilers can help
  - Often research compilers ;)

***Based on:**

M. Wolf, M. Lam, *A Data Locality Optimizing Algorithm*. ACM SIGPLAN, PLDI (1991)

Loop representation as Iteration Space

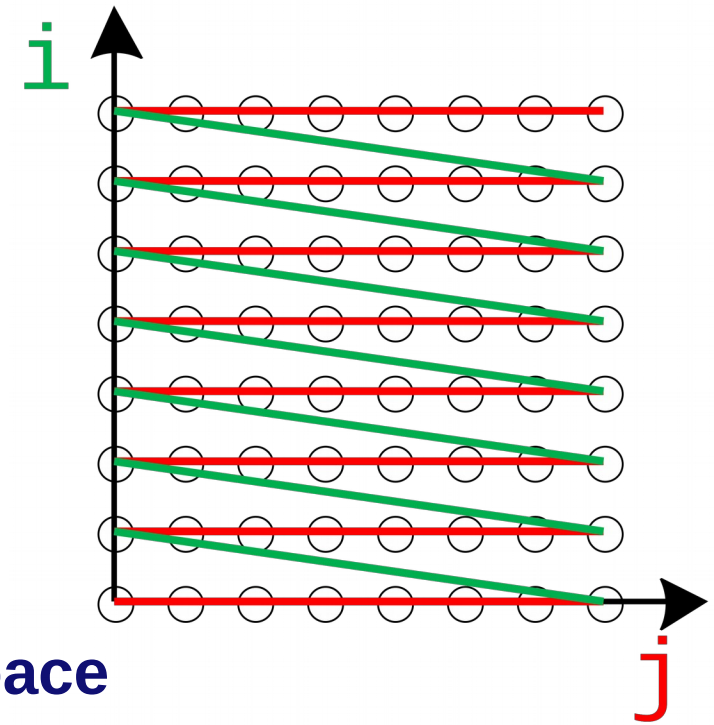

```
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    A[i][j] = B[j][i];
```



- **Each position represents an iteration**

Visitation order in Iteration Space


```
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    A[i][j] = B[j][i];
```

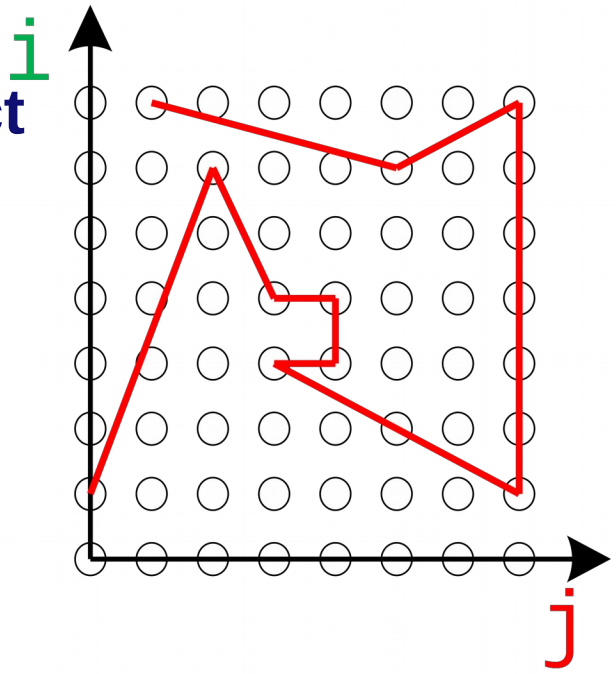


- **Iteration space is not data space**

But we don’t have to be regular


```
A[j][j] = B[j][i];
```

- But order can have huge impact
- Memory Hierarchy
  - E.g. Reuse of data
- Parallelism
- Loop or control overhead

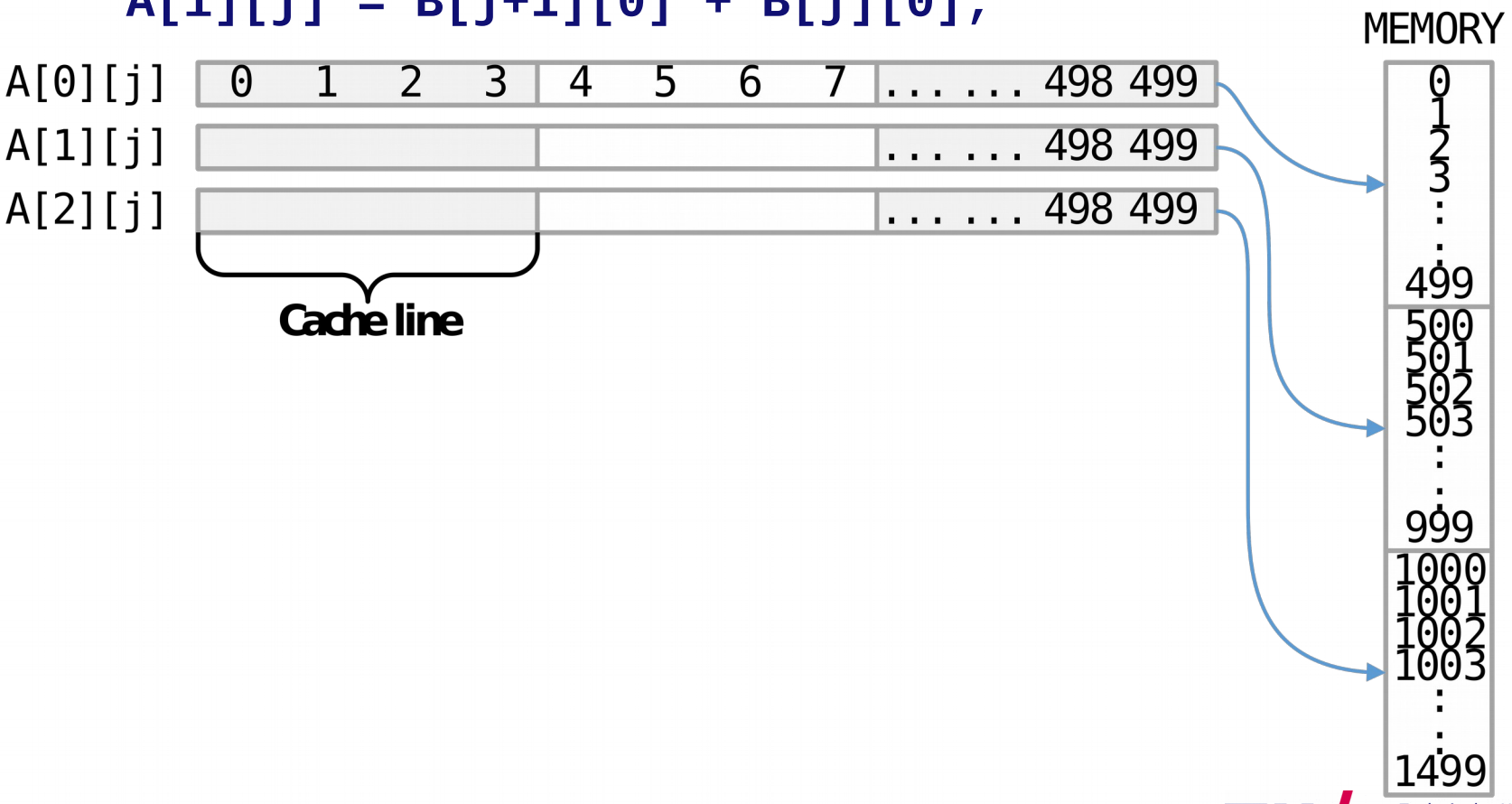


Types of data reuse

Assumptions:

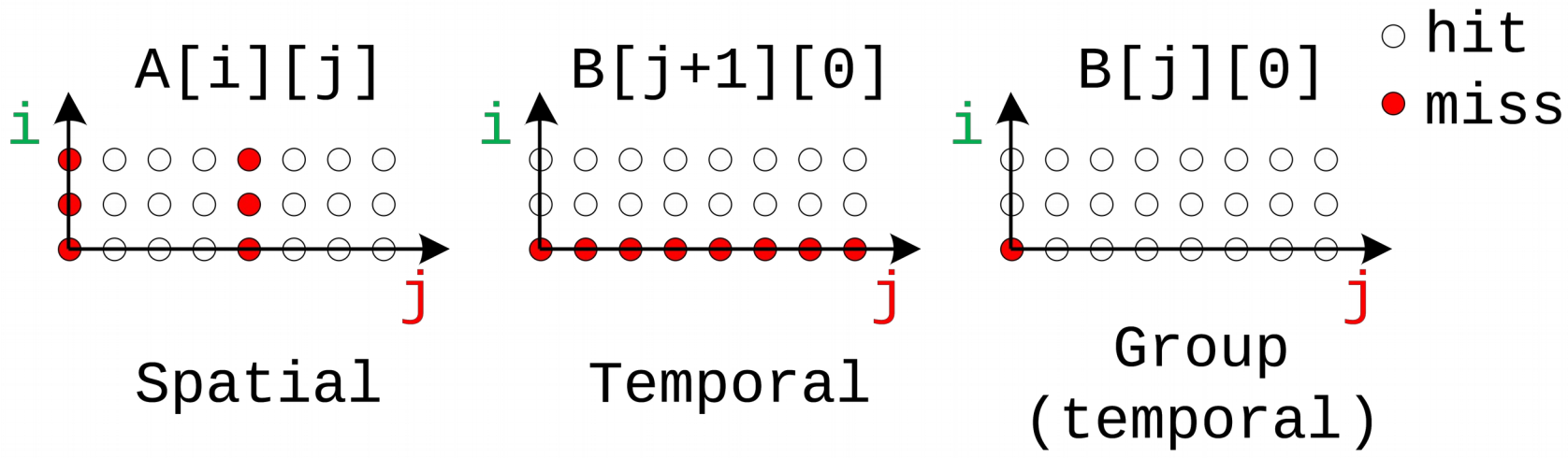
1. Memory layout row mayor
2.Cache line size=4

```
for i := 0 to 2
  for j := 0 to 100
    A[i][j] = B[j+1][0] + B[j][0];
```



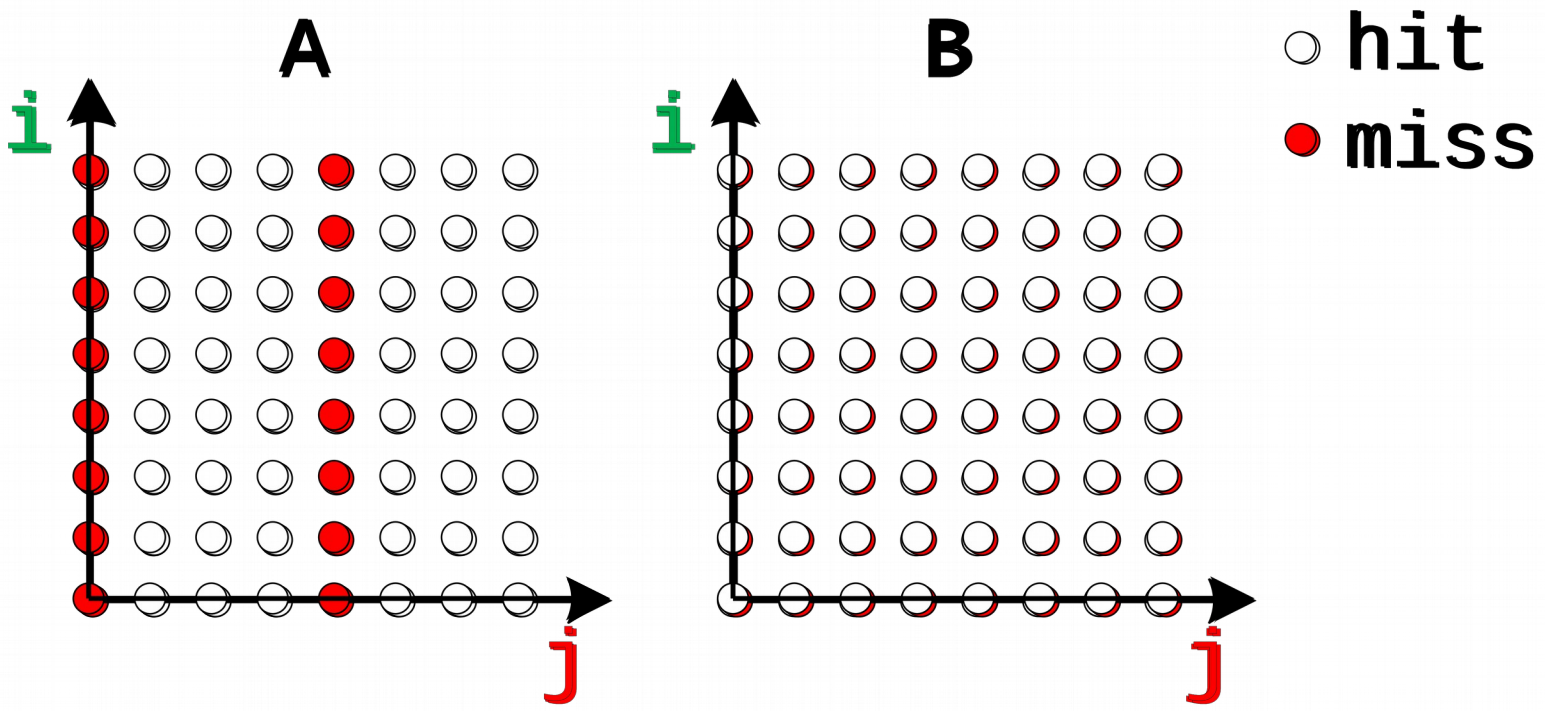
Types of data reuse


```
for i := 0 to 2
  for j := 0 to 100
    A[i][j] = B[j+1][0] + B[j][0];
```



When do cache misses occur?


```
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    A[i][j] = B[j][i];
```



Optimize array accesses for memory hierarchy

- Solve the following questions?
  - When do misses occur?
    - Use “locality analysis”
  - Is it possible to change iteration order to produce better behavior?
    - Evaluate the cost of various alternatives
  - Does the new ordering produce correct results?
    - Use “dependence analysis”

Different Reordering Transformations

- Loop Interchange
- Tiling

} Can improve locality

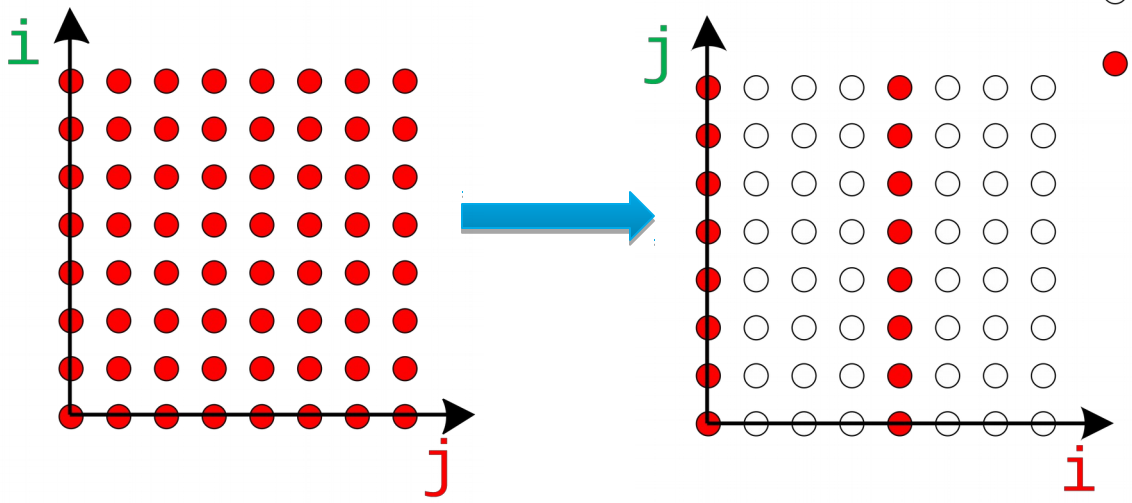
- Skewing
- Loop Reversal

} Can enable above

Loop Interchange or Loop Permutation

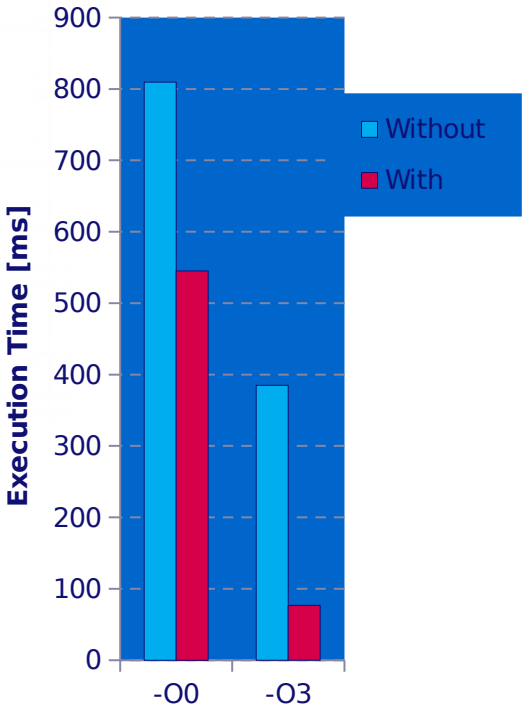

```
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    A[j][i] = i*j;
```

```
for(j=0; j<N; j++){
  for(i=0; i<N; i++){
    A[j][i] = i*j;
```



○ hit

● miss



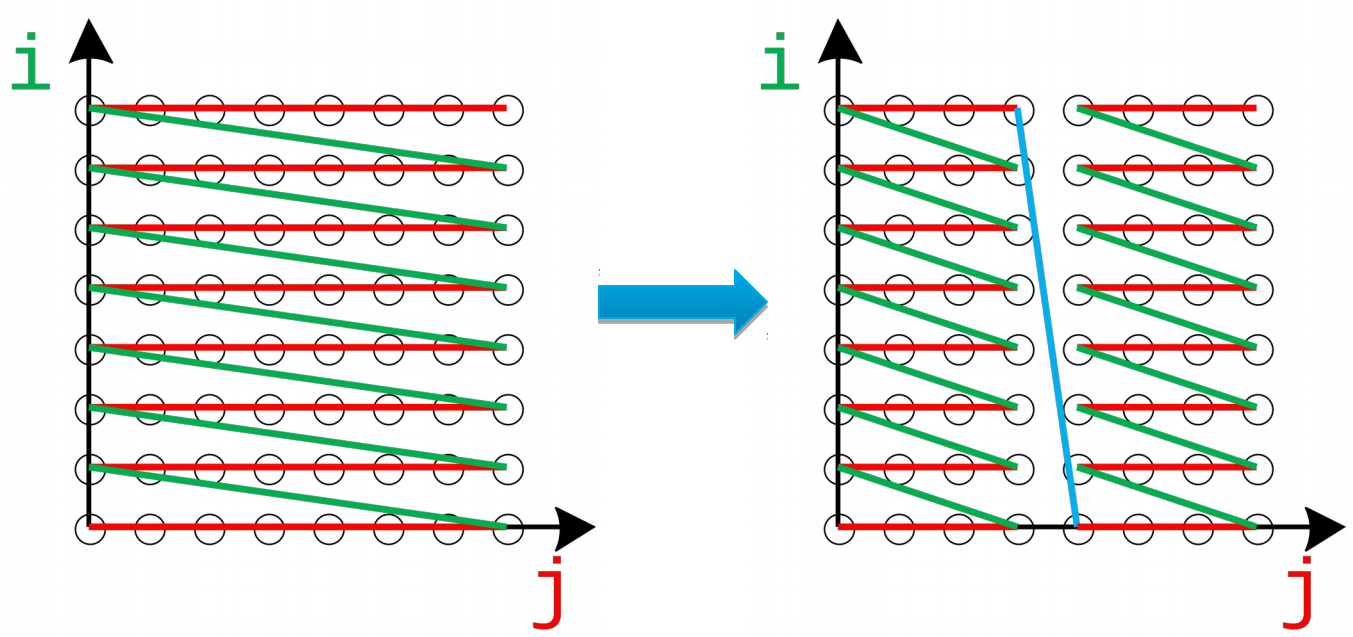
- N is large with respect to cache size
- Normal compilers will not help you!

Tiling the Iteration Order

- **Result in Iteration Space**

```
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    f(A[i],A[j]);
```

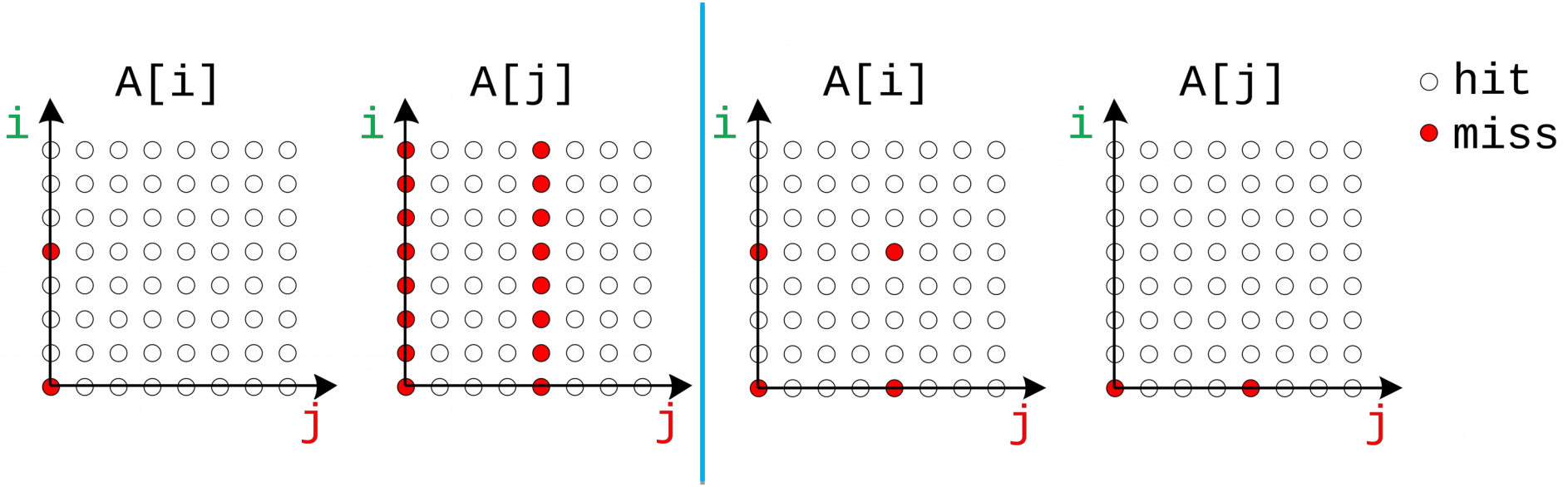
```
for(jj=0; jj<N; jj+=Tj){
  for(i=0; i<N; i++){
    for(j=jj; j<jj+Tj; j++){
      f(A[i],A[j]);
```



Cache Effects of Tiling


```
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    f(A[i],A[j]);
```

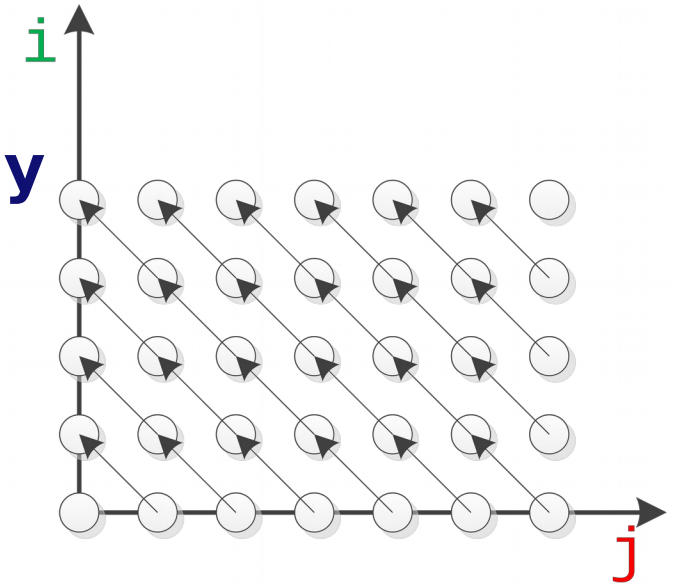
```
for(jj=0; jj<N; jj+=Tj){
  for(i=0; i<N; i++){
    for(j=jj; j<jj+Tj; j++){
      f(A[i],A[j]);
```



Is this always possible?

- No, dependencies can prevent transformations
- Interchange and Tiling is not always valid!
- Example Interchange
  - Spatial locality
  - Dependency (1,-1)
- Should be lexicographically correct

```
for(i=1; i<N; i++){
  for(j=0; j<N; j++){
    a[j][i] = a[j+1][i-1] + 1;
  }
}
```



Lexicographical ordering

- **Each iteration is a node identified by its index vector: p** = $(p_1, p_2, \ldots, p_n)$
- **Iterations are executed in lexicographic order**
- **For p** = $(p_1, p_2, \ldots, p_n)$ **and q** = $(q_1, q_2, \ldots, q_n)$ f
- **p is lexicographically greater than q written as p q**
- **Therefore iteration p must be executed after q**
- **Example:**

  (1,1,1), (1,1,2), (1,1,3), …,

  (1,2,1), f (1,2,2), (1,2,3), … p

  …,

  (2,1,1), (2,1,2), (2,1,3), …,

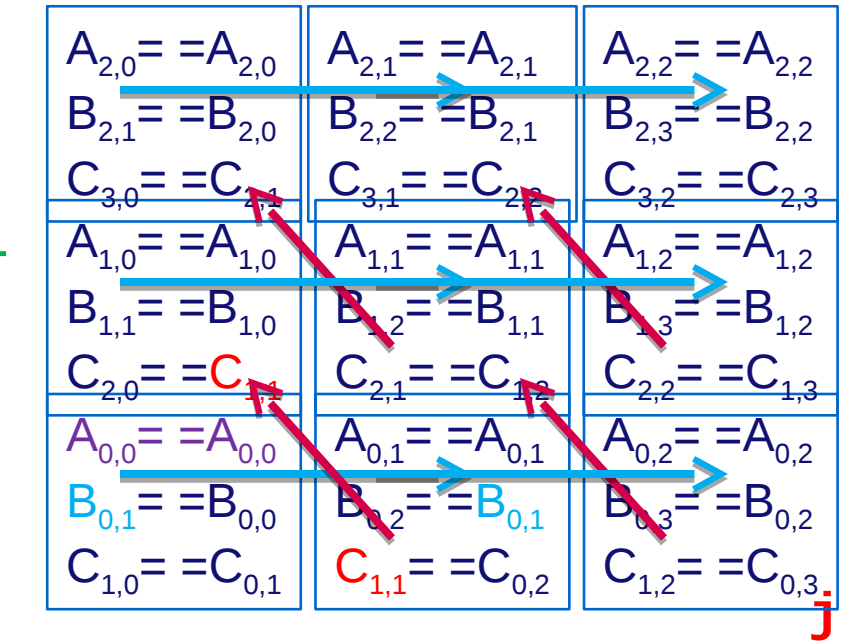
A little quiz?

Dependence vectors

- **Dependence vector in an n-nested loop is denoted as a vector: d = (d_1, d_2, …, d_n)**
- **A single dependence vector represents a set of distance vectors**
- **A distance vector defines a distance in the iteration space**
- **Distance vector is difference between target and source iterations:**
  - **$d = I_T - I_S$**
- **The distance of the dependence is:**
  - **$I_S + d = I_T$**

Example of dependence vector


```
For(i=0; i<N; i++){
  for(j=0; j<N; j++)
    A[i,j] = …;
         = A[i,j];
    B[i,j+1] = …;
         = B[i,j];
    C[i+1,j] = …;
         = C[i,j+1];
```



A yields:

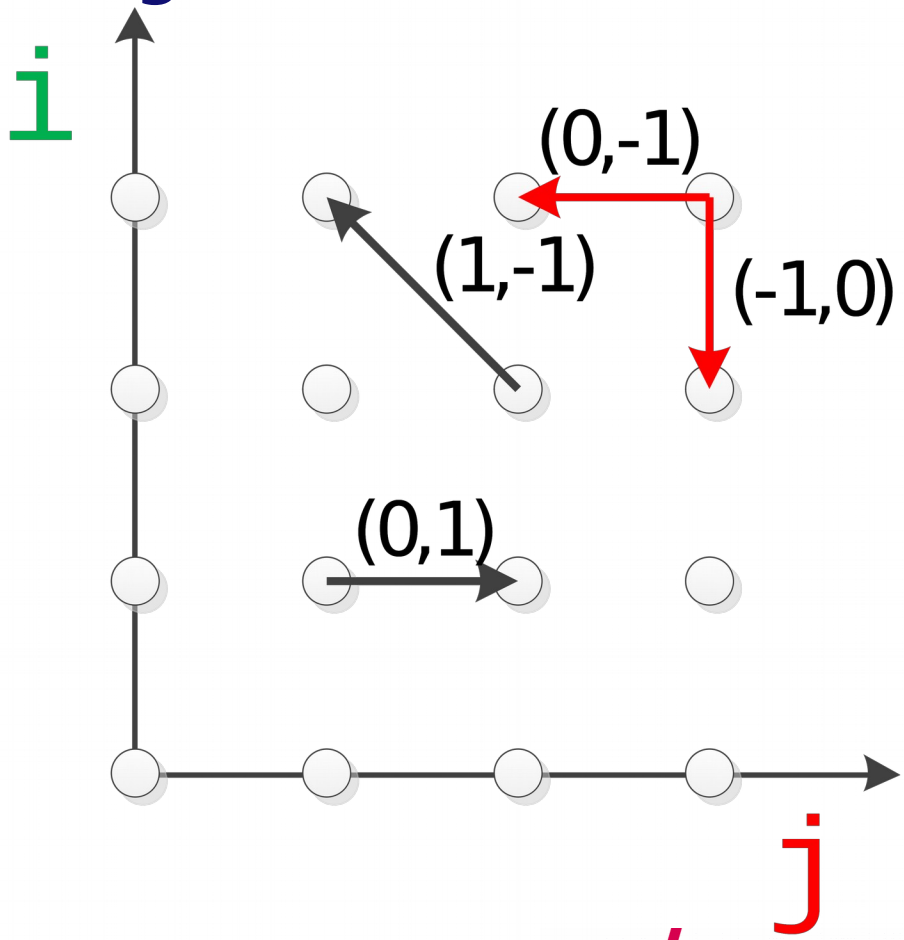
yields:

B yields:

Plausible dependence vectors

- **A dependence vector is plausible iff it is lexicographically non-negative**

- **Plausible: (1,-1)**
- **Implausible: (-1,0)**



Loop Reversal

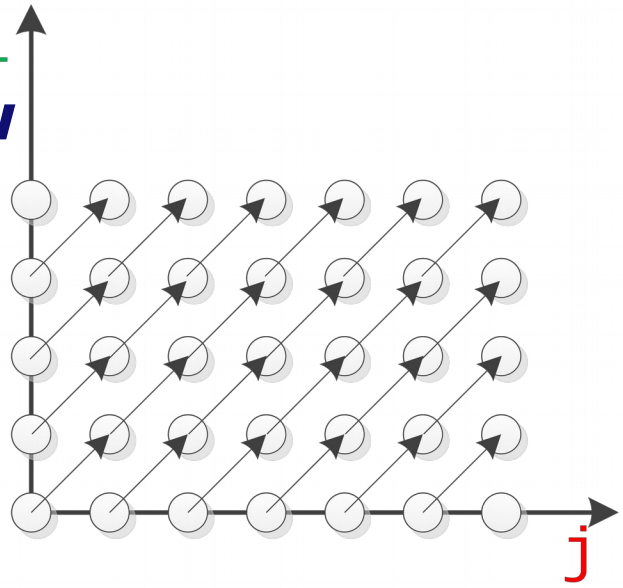
- **Dependency (1,-1)**

```
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
    a[j][i] = a[j+1][i-1] + 1;
  }
}
```

- **Interchange is valid now**

Dependency (1,1)

```
for(i=0; i<N; i++){
  for(j=N-1; j>=0; j--){
    a[j][i] = a[j+1][i-1] + 1;
  }
}
```

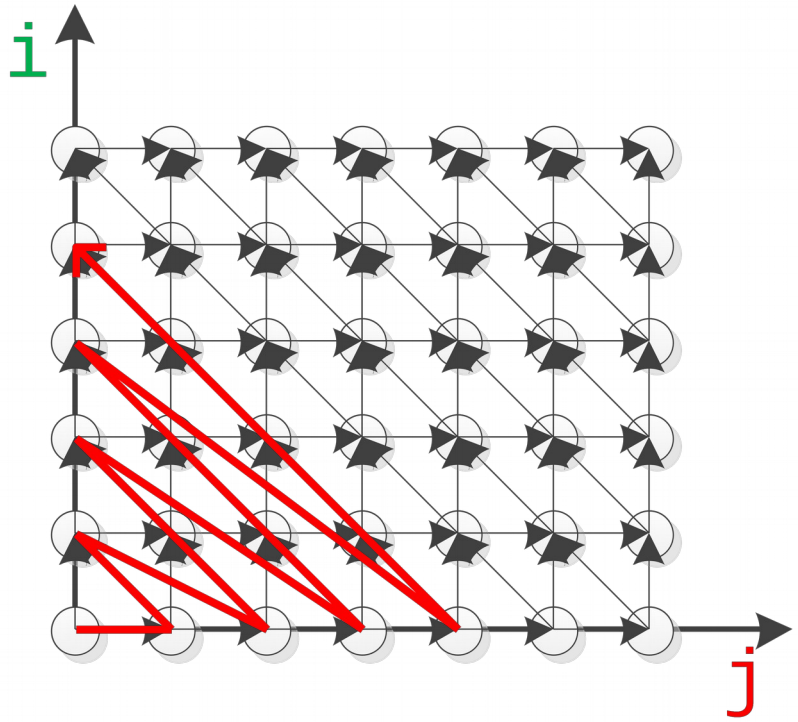


More difficult example


```
For(i=0; i<6; i++){
  for(j=0; j<7; j++){
    A[j+1] += 1/3 * (A[j] + A[j+1] + A[j+2]);
```

D={(0,1),(1,0),(1,-1)}

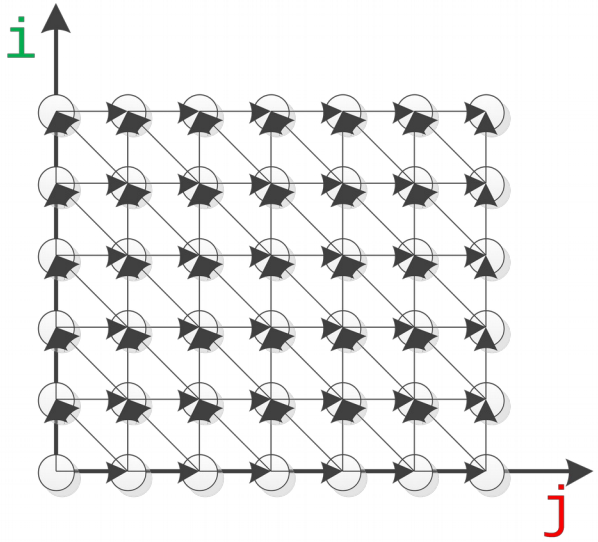
- **What are other legal visit orders?**

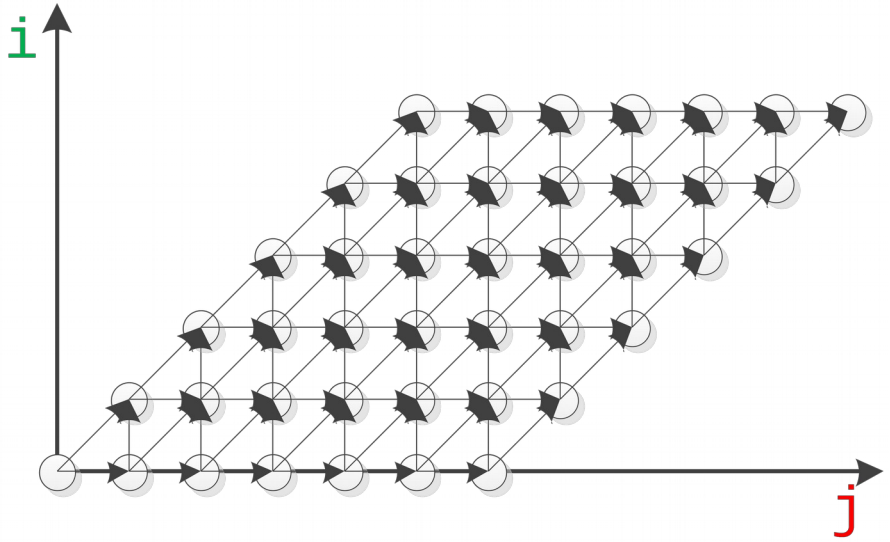


Loop Skewing


```
for(i=0 i<6; i++){
  for(j=0; j<7; j++){
    A[j+1] = 1/3 * (A[j] + A[j+1] + A[j+2]);
```

D = {(0,1),(1,0),(1,-1)}



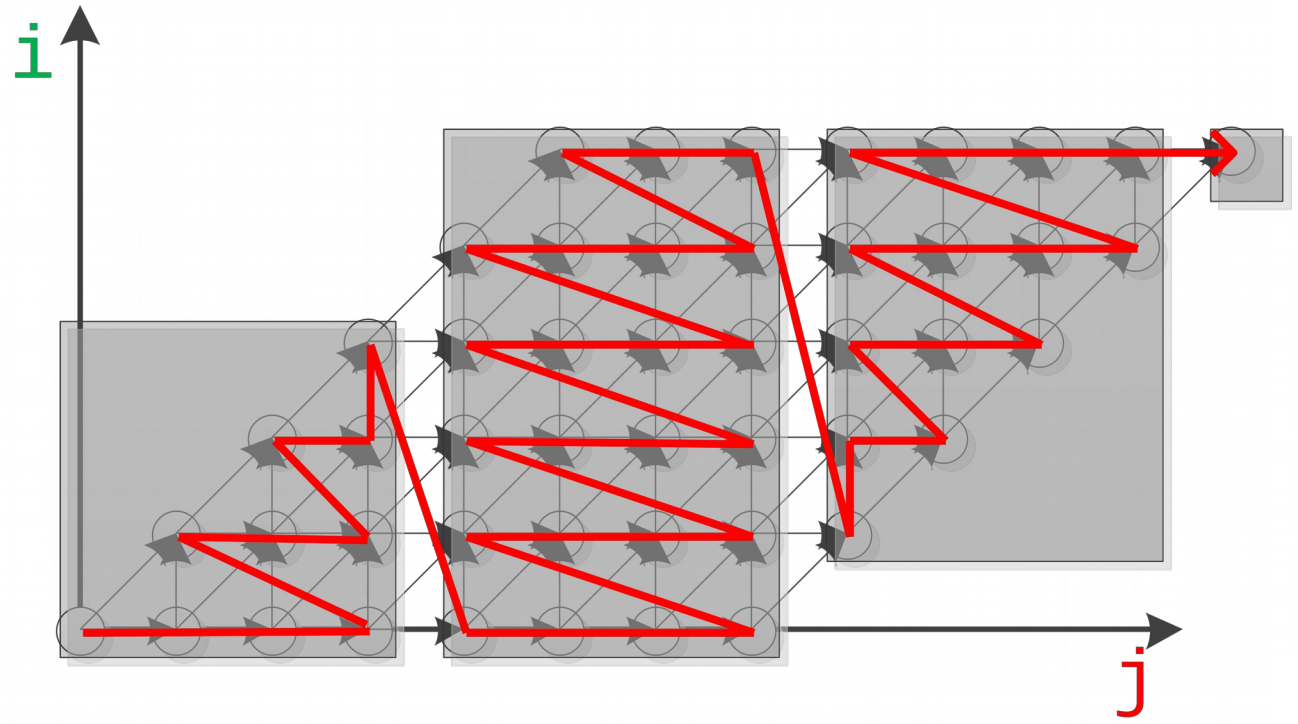



```
for(i=0; i<6; i++){
  for(j=i; j<7+i; j++){
    A[j-i+1] = 1/3 * (A[j-i] + A[j-i+1] + A[j-i+2]);
```

D = {(0,1),(1,1),(1,0)}

Enables Tiling

- **Apply Skew Transformation**
- **Apply Tiling**



Is It Really That Easy ?

- No, we forget a few things
- Cache capacity and reuse distance
  - If caches were infinitely large, we would be finished
- For deep loopnests analysis can be very difficult
- It would be much better to automate these steps
- We need more the tools

Cost mode ls


```
for(y=0; y<By; y++){
  for(x=0; x<Bx; x++){
    for(k=0; k<Bk; k++){
      for(l=0; l<Bl; l++){
        out[y][x] += in[y+k][x+l] * c[k][l];
```

Reuse distance

- **The number of unique accesses between a use and a reuse is defined as the reuse distance**

  [a[0] a[6] a[20] b[1] a[6] a[0]] $R_D = 3$

*** C. Ding, Y. Zhong, *Predicting Whole-Program Locality through Reuse Distance Analysis*, PLDI (2003)**

- **If fully associative cache with n=4 entries**
- **Optimal least recently used replacement policy**
- **Cache hit if $R_D < n$**
- **Cache miss if $R_D \geq n$**

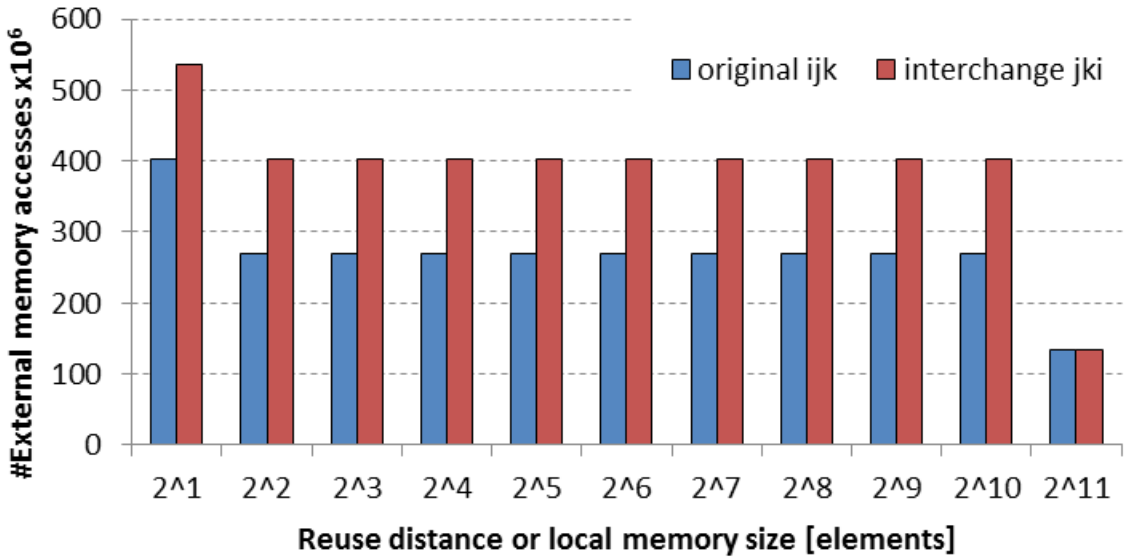
- **We can measure reuse distance with profiling tools:**

Study a matrix multiplication kernel

- **Loop interchange**

```
for(i=0; i<Bi; i++){
  for(j=0; j<Bj; j++){
    for(k=0; k<Bk; k++){
      C[i][j] += A[i][k] * B[k][j];
```

```
for(j=0; j<Bj; j++){
  for(k=0; k<Bk; k++){
    for(i=0; i<Bi; i++){
      C[i][j] += A[i][k] * B[k][j];
```

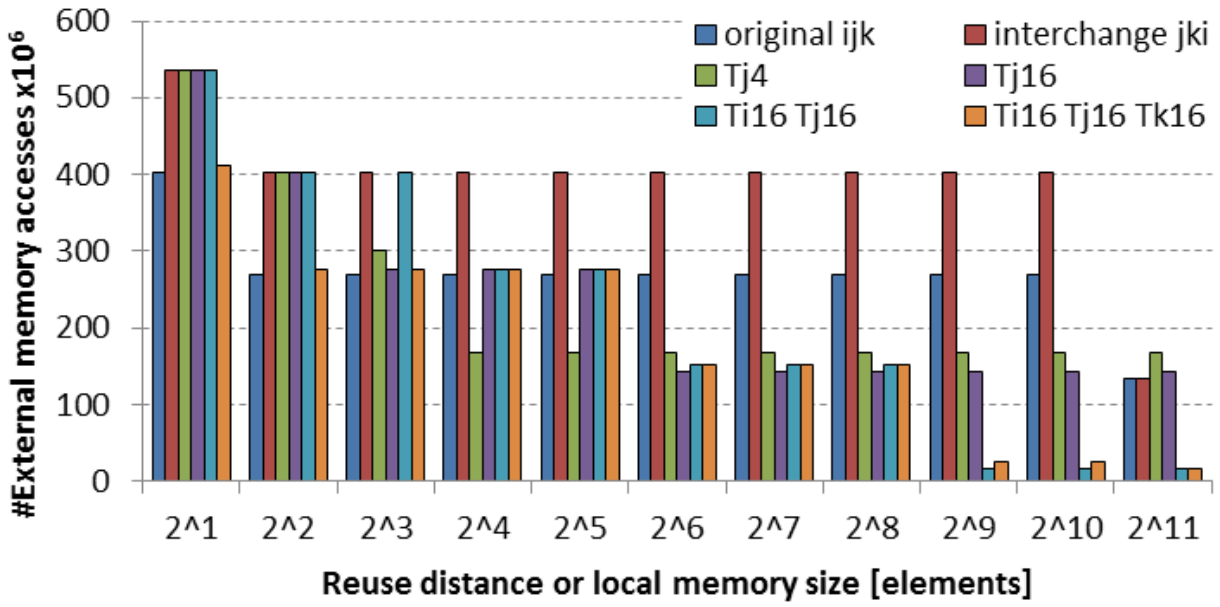


Loop tiling


```
for(i=0; i<Bi; i++){
  for(j=0; j<Bj; j++){
    for(k=0; k<Bk; k++){
      C[i][j] += A[i][k] * B[k][j];
```

```
for(i=0; i<Bi; i++){
  for(jj=0; jj<Bj; jj+=Tj){
    for(k=0; k<Bk; k++){
      for(j=jj; j<jj+Tj; j++)
        C[i][j] += A[i][k] * B[k][j];
```

- **Can be done in multiple dimensions**
- **Manual exploration is intractable**

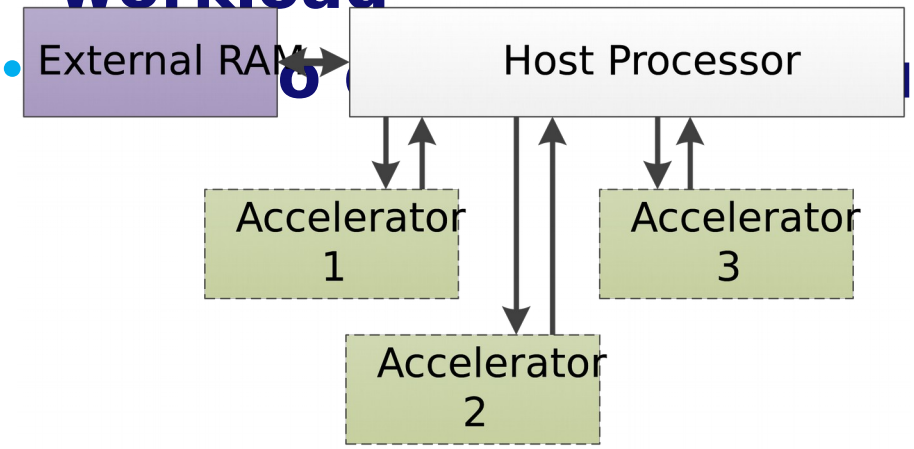


Automate the search

“Never send a human to do a machine’s job.” Agent Smith, The Matrix (1999)

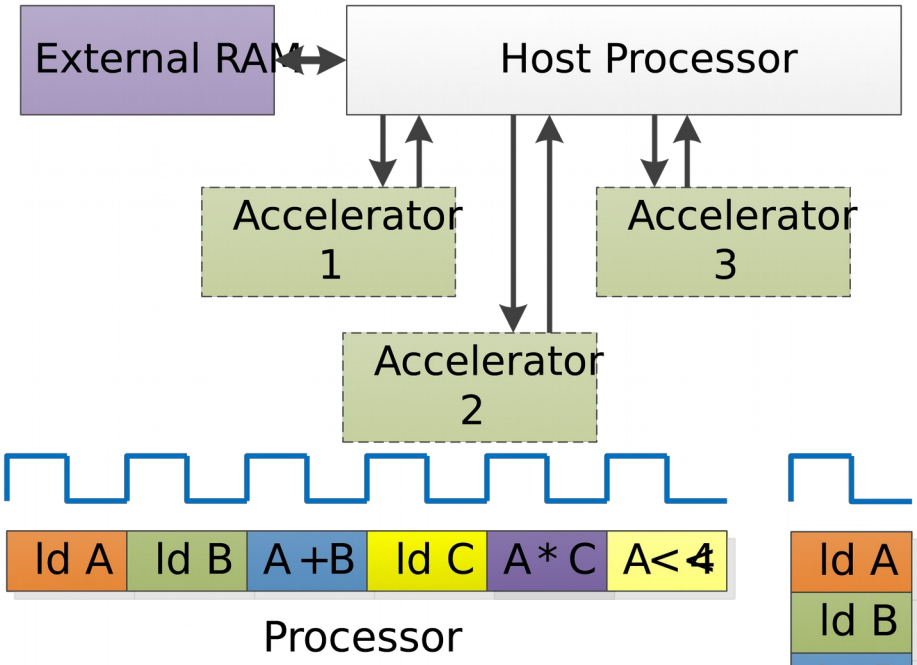
A Practical Intermezzo

- **Accelerators for energy efficiency**
- **Special Purpose HW**
  - **~100x better performance, 100x less energy than CPU**
- **A simple processor for control**
- **Multiple accelerators for heavy compute workload**
- **Co[illegible]rrent HLS languages**

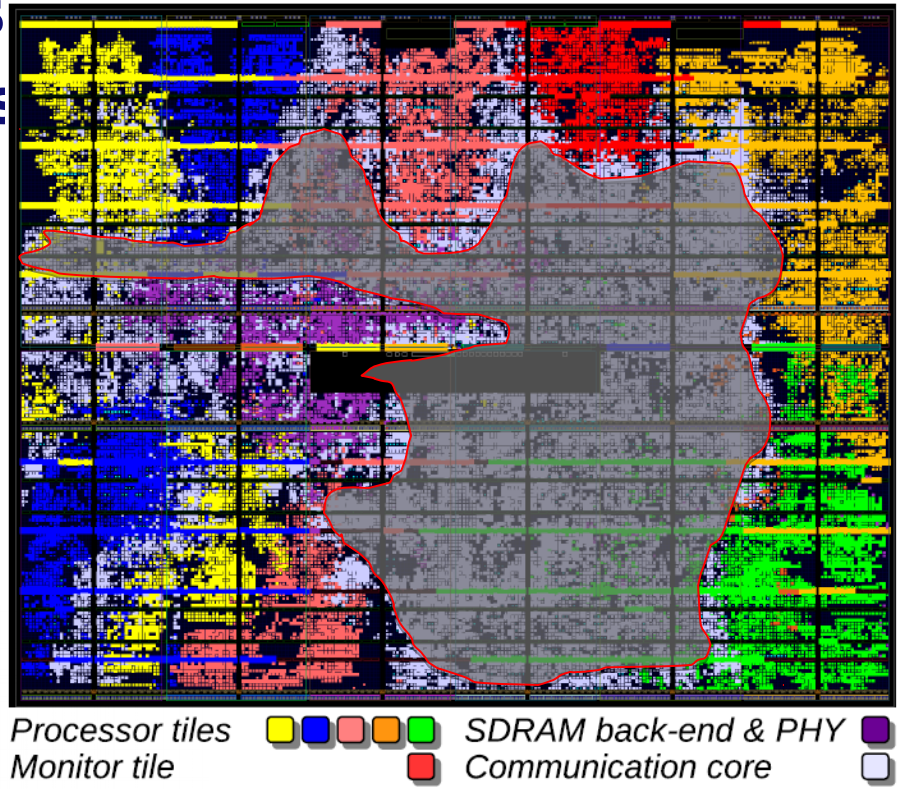


Bandwidth Hunger

- **Simple RISC core**
- **Accelerator Operations**
- **Interconnect Resources**

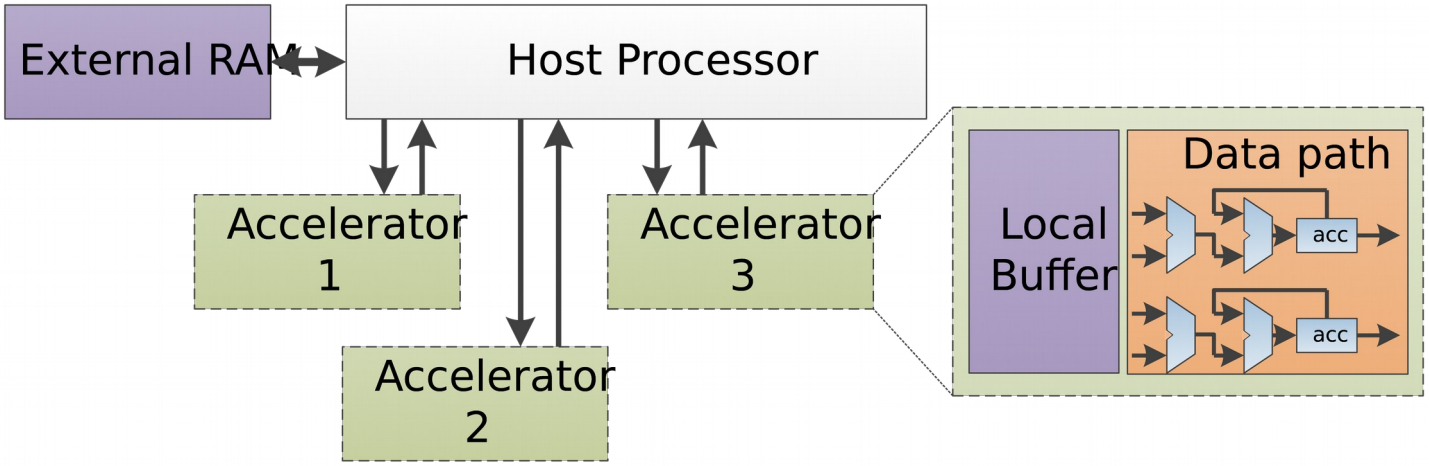


Aethereal Interconnect 6 cores



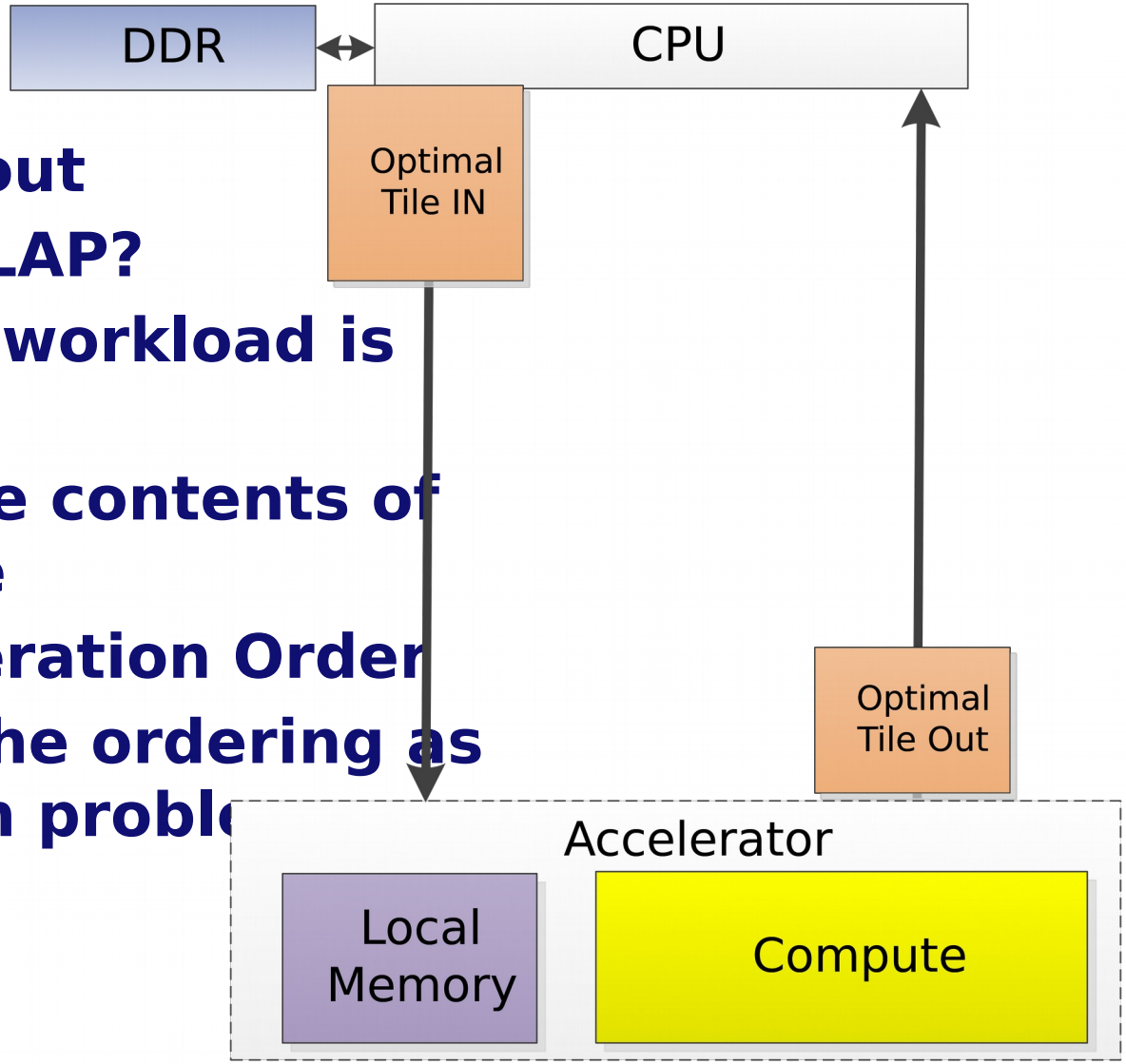
What can we do?

- **Exploit the data reuse in accelerator workload**
- **Use local buffers**
- **Reduce communication with external memory**



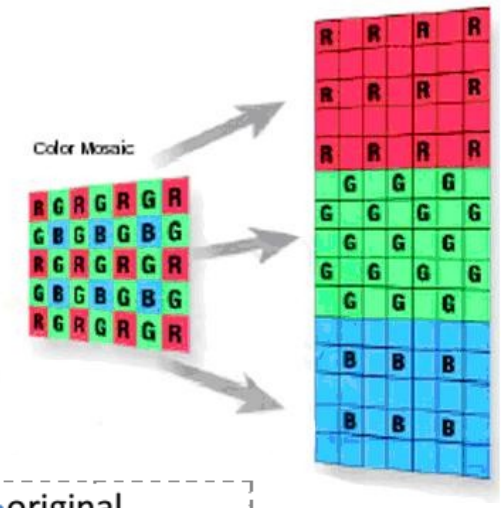
A Different Approach: Inter-Tile Reuse

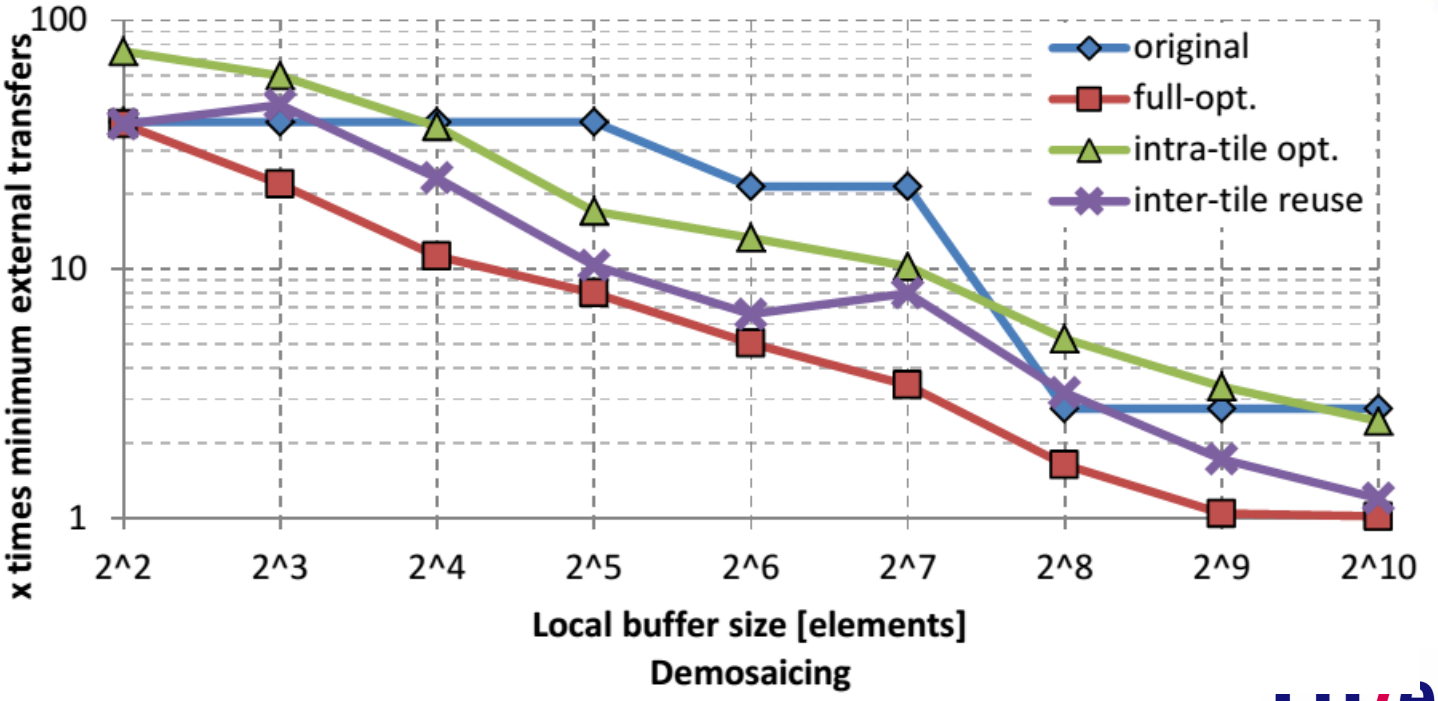
- But how about
  DATA OVERLAP?
- Accelerator workload is often static
- We know the contents of the next tile
- Optimize Iteration Order
- Formulate the ordering as optimization proble



Demosaic Benchmark

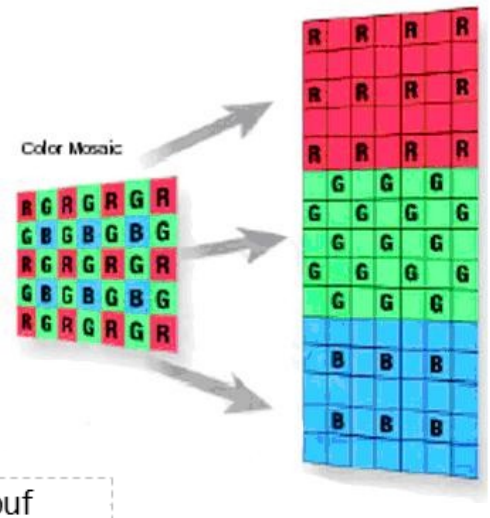
- **Convert an 8 Mpix image: 0.6 G**
  - **Intel-i7 0.56 s**
  - **ARM-A9 5.75 s**
  - **Microblaze 22.10 s**
  - **Accelerator 1.35 s @ 50 M**

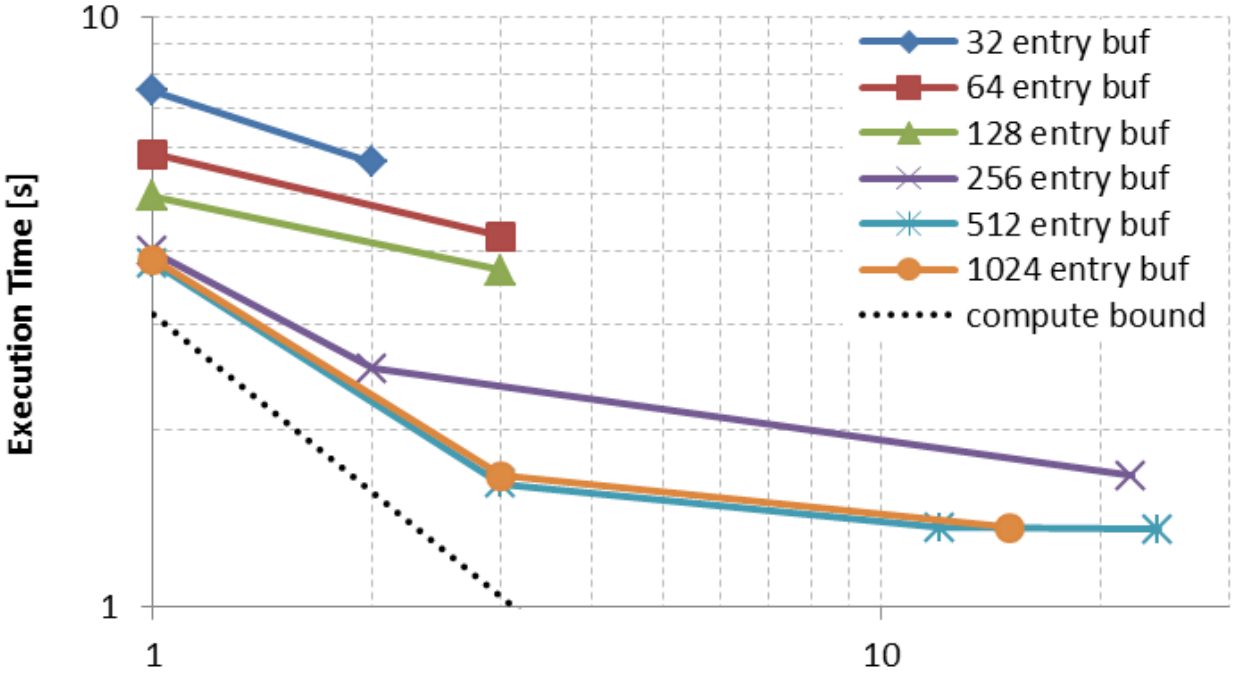




Demosaic Benchmark

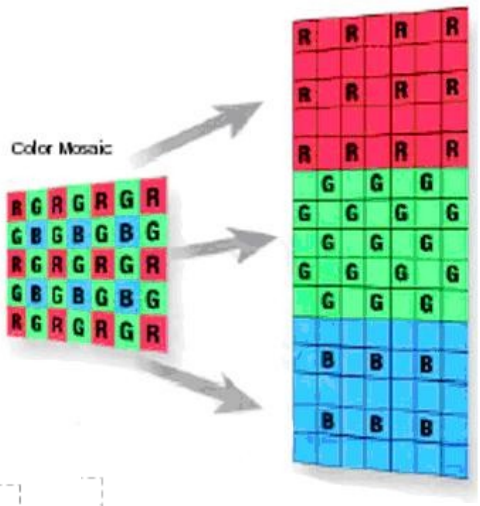
- **Convert an 8 Mpix image: 0.6 G**
  - **Intel-i7 0.56 s**
  - **ARM-A9 5.75 s**
  - **Microblaze 22.10 s**
  - **Accelerator 1.35 s @ 50 M**

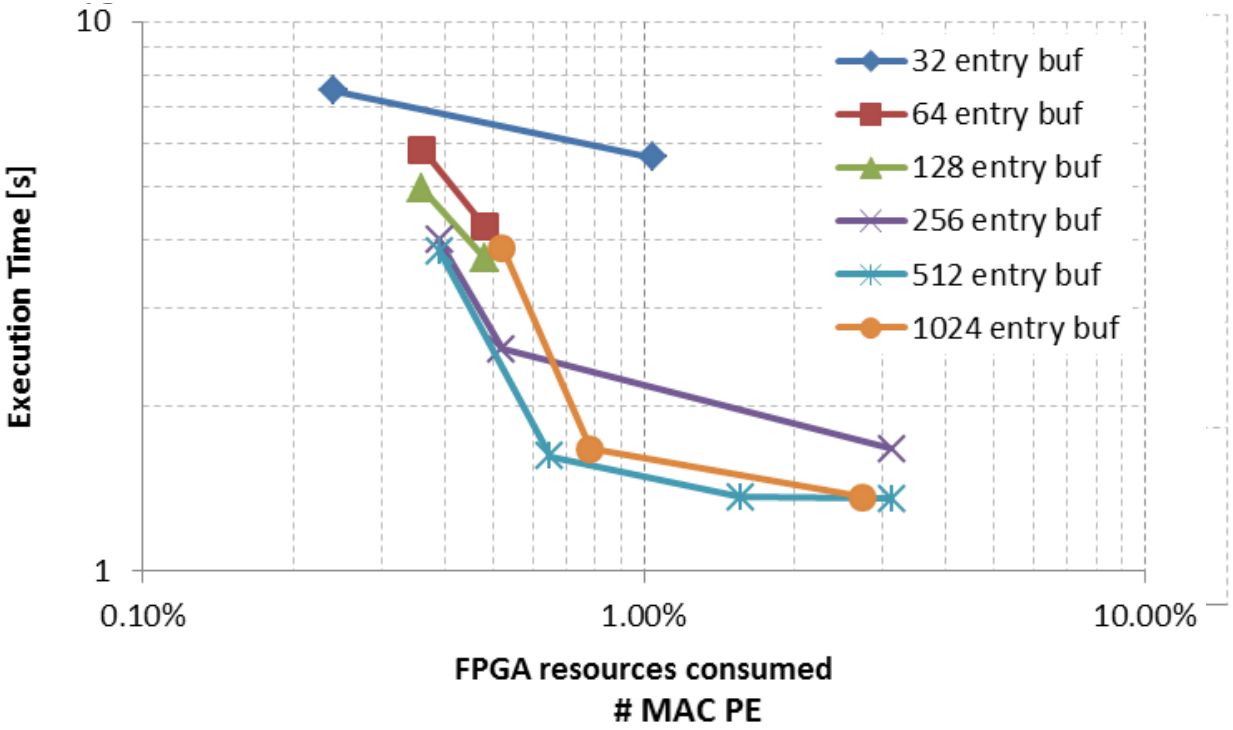




Demosaic Benchmark

- **Convert an 8 Mpix image: 0.6 G**
  - **Intel-i7 0.56 s**
  - **ARM-A9 5.75 s**
  - **Microblaze 22.10 s**
  - **Accelerator 1.35 s @ 50 M**

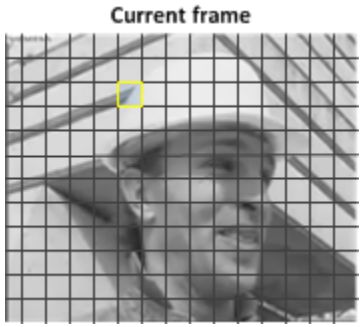


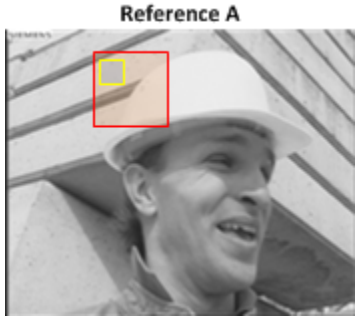


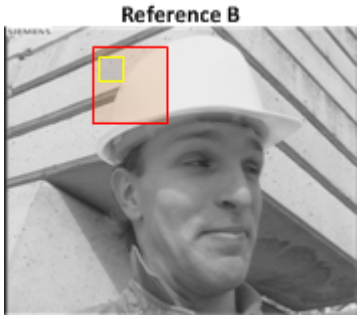
Motion Estimation

- **H264 video encoder ME ~ 30 Gops**

```
for(frame){
  for(macroblocks){
    for(reference frame){
      for(searchrange){
        for(pixel){
          SAD_operation();
    for(find max){
    Store vector();
```

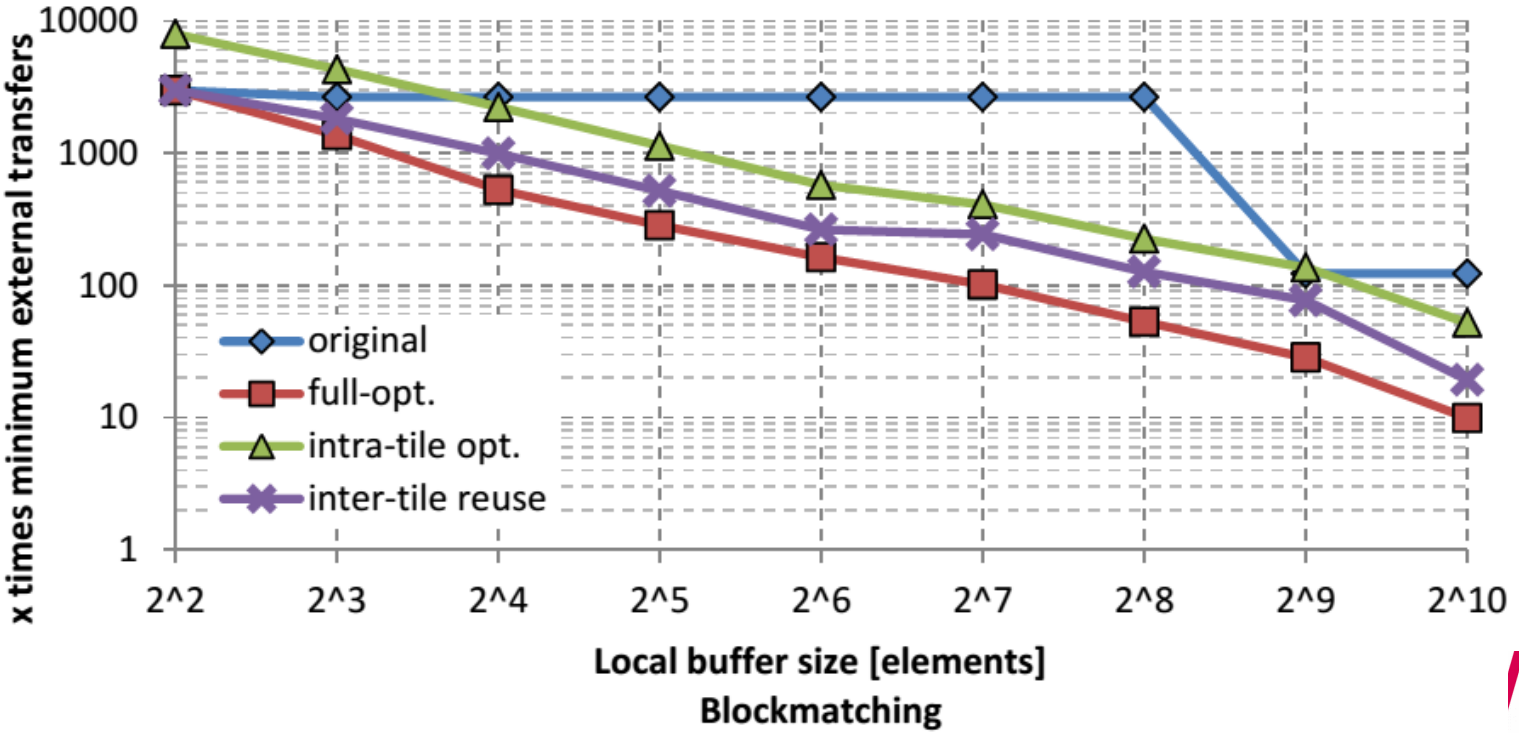






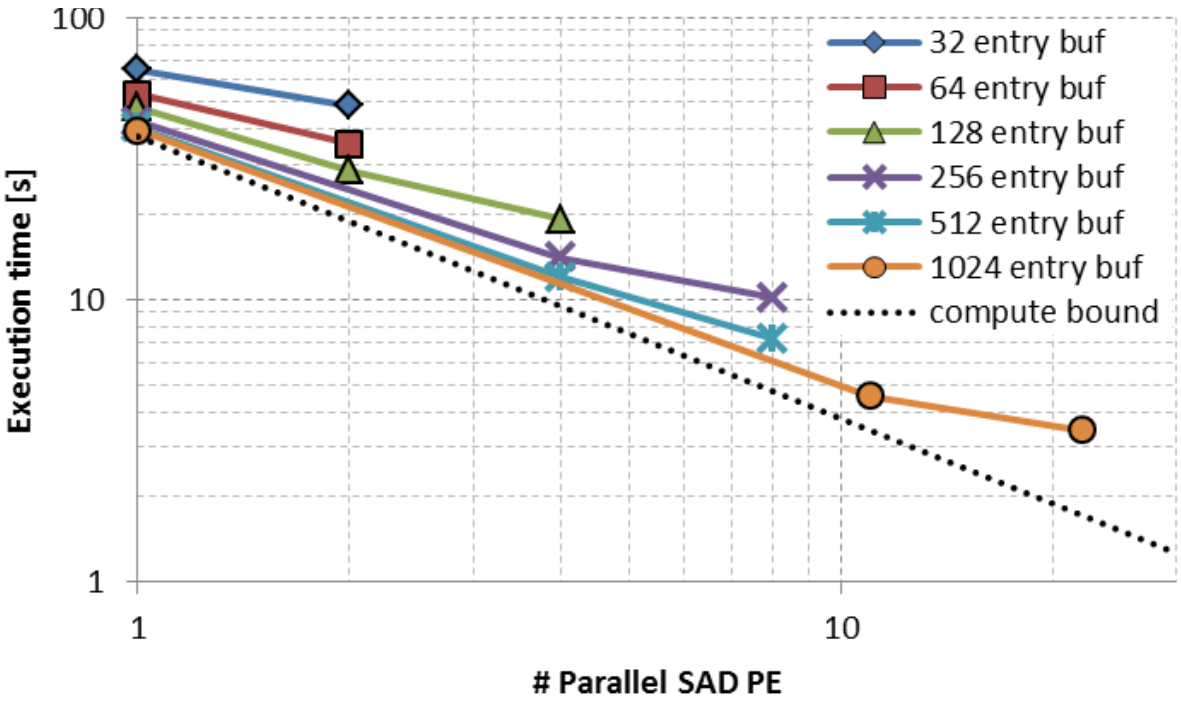
Block Matching

- **H264 video encoder ME ~ 30 Gops**
  - **Intel-i7 8.1 sec**
  - **ARM A9 72.3 sec**
  - **Microblaze 283.9 sec**
  - **Accelerator 3.4 sec @ 50MB/s**

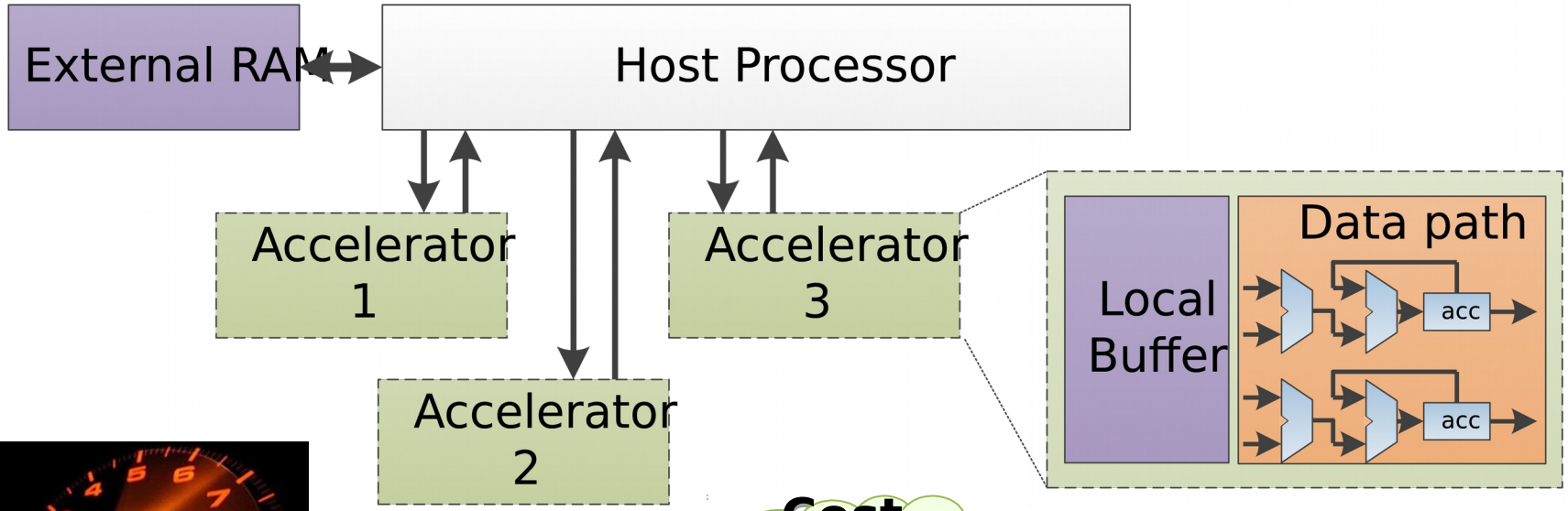


Block Matching

- **H264 video encoder ME ~ 30 Gops**
  - **Intel-i7 8.1 sec**
  - **ARM A9 72.3 sec**
  - **Microblaze 283.9 sec**
  - **Accelerator 3.4 sec @ 50MB/s**



What do we achieve



Cost models

```
for(y=0; y<By; y++){
  for(x=0; x<Bx; x++){
    for(k=0; k<Bk; k++){
      for(l=0; l<Bl; l++){
        out[y][x] += in[y+k][x+l] * c[k][l];
```

Loop Transformation Classes

- **Data-Flow-Based Transformations**
- **Iteration Reordering Transformations**
- **Loop Restructuring Transformations**
  - **Alter the form of the loop**

Loop Unrolling

- Duplicate loop body and adjus loop header
  - Increases ILP
  - Reduces loop overhead
  - Possibilities for common subexpression elimination
- If partial unroll, loop should stay in original bound!
- Downsides
  - Code size increases, can increase I-cache miss-rate

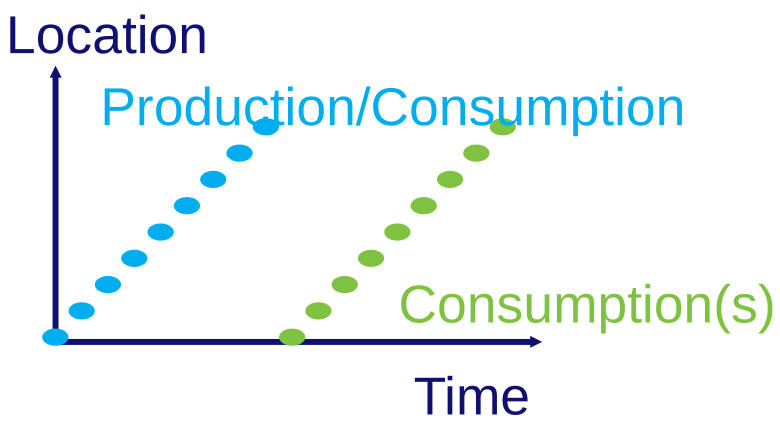
```
for(i=1; i<N; i++){
  a[i]=a[i-1]+a[i]+a[i+1];
}
```

```
for(i=1; i<N-1; i+=2){
  a[i]=a[i-1]+a[i]+a[i+1];
  a[i+1]=a[i]+a[i+1]+a[i+2];
}
if(mod(N-1,2)==1){
  a[N-1]=a[N-2]+a[N-1]+a[N];
```

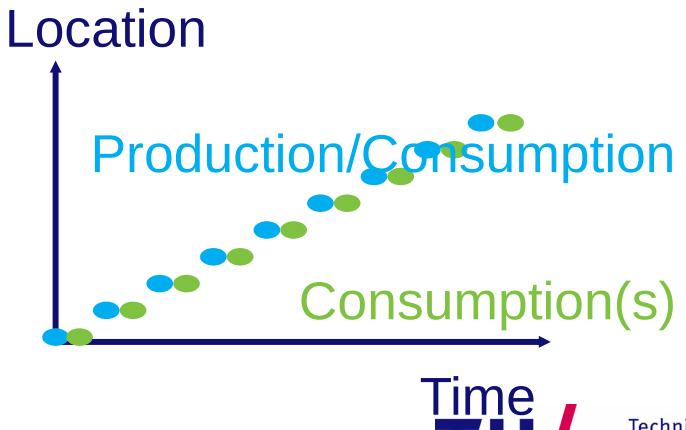
Loop Merge or Fusion

- **Why?**
  - **Improve locality**
  - **Reduce loop overhead**

```
for(i=0; i<N; i++){
  b[i]=a[i]+1;
}
for(j=0; j<N; j++){
  c[j]=b[j]+a[j];
}
```



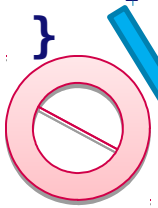

```
for(i=0; i<N; i++){
  b[i]=a[i]+1;
  c[j]=b[j]+a[j];
}
```



Merging is not always allowed

- Data dependencies from first to second loop
- Allowed if
  - ∀ I: cons(I) in loop 2 ≤ prod(I) in loop 1
- Enablers: Bump, Reverse, Skew

```
for(i=0; i<N; i++){
  b[i]=a[i]+1;
}
```



N-1>=i

```
for(i=0; i<N; i++){
  c[i]=b[N-1]+a[i];
}
```

```
for(i=0; i<N; i++){
  b[i]=a[i]+1;
}
```

i-2 < i

```
for(i=0; i<N; i++){
  c[i]=b[i-2]+a[i];
}
```

Loop Bump: Example as enabler


```
for(i=2; i<N; i++){
  b[i]=a[i]+1;
}
for(i=0; i<N-2; i++){
  c[i]=b[i+2]*0.5;
}
```

i+2 -> i => direct merging not possible

```
for(i=2; i<N; i++){
  b[i]=a[i]+1;
}
```

i+2-2 = i => possible to merge

```
for(i=2; i<N; i++){
  c[i-2]=b[i+2-2]*0.5;
}
```

```
for(i=2; i<N; i++){
  b[i]=a[i]+1;
  c[i-2]=b[i]*0.5;
}
```

Automatic frameworks

- These are the tools to abstract away from code:
- Model reuse (potential for locality)
- Local iteration space

- Transform loops to take advantage of reuse
- Does this still give valid results?

- Loop transformation theory
  - Iteration space
  - Dependence vectors
  - Unimodular transformations
  - Polyhedral Model

Tricks of the Trade: Main Take-Aways

- Data-Flow-Based Transformations
  - Good compilers can help
- Iteration Reordering Transformations
  - High-level transformations
  - Compilers are often not enough
  - Huge gains for parallelims and locality
  - Be carefull with dependencies
- Loop Restructuring Transformations
  - Similar to reordering
- Tools can help you: Check out SLO
- Optimizing source to source compilers as well
  - If you have luck, these don’t support all constructions

Extra Enabler Transformations

- Many of these can be very usefull
- Especially to enable the bennificial ones:
  - Interchange
  - Tiling
  - Fusion
- Often you need combinations to get the job done!

- Make sure you can use them all for the:
  Data Memory Management Assingment

Loop Extend

for I = exp_1 to exp_2
    A(I)

$$\begin{pmatrix} exp_3 \leq exp_1 \\ exp_4 \geq exp_2 \end{pmatrix}$$

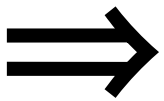
$\Rightarrow$

for I = exp_3 to exp_4
    if I≥exp_1 **and** I<exp_2
        A(I)

Loop Extend: Example as enabler

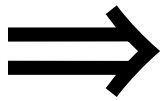
```
for (i=0; i<N; i++)
   B[i] = f(A[i]);
for (i=2; i<N+2; i++)
   C[i-2] = g(B[i]);
```

Loop Extend



```
for (i=0; i<N+2; i++)
   if(i<N)
     B[i] = f(A[i]);
for (i=0; i<N+2; i++)
   if(i>=2)
      C[i-2] = g(B[i]);
```

Loop Merge



```
for (i=0; i<N+2; i++)
   if(i<N)
     B[i] = f(A[i]);
   if(i>=2)
      C[i-2] = g(B[i]);
```

Loop Reduce

for I = exp_1 to exp_2
  if I≥exp_3 **and** I<exp_4
    A(I)

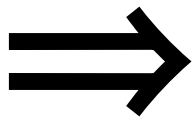
⇒

for I = max(exp_1,exp_3) to
        min(exp_2,exp_4)
  A(I)

Loop Body Split

for I = exp_1 to exp_2
    A(I)
    B(I)

A(I) must be single-assignment: its elements should be written once



for I_a = exp_1 to exp_2
    $A(I_a)$

for I_b = exp_1 to exp_2
    $B(I_b)$

Loop Body Split: Example as enabler

```
for (i=0; i<N; i++)
 A[i] = f(A[i-1]);
 B[i] = g(in[i]);
for (j=0; j<N; j++)
 C[j] = h(B[j],A[N]);
```

Loop Body Split

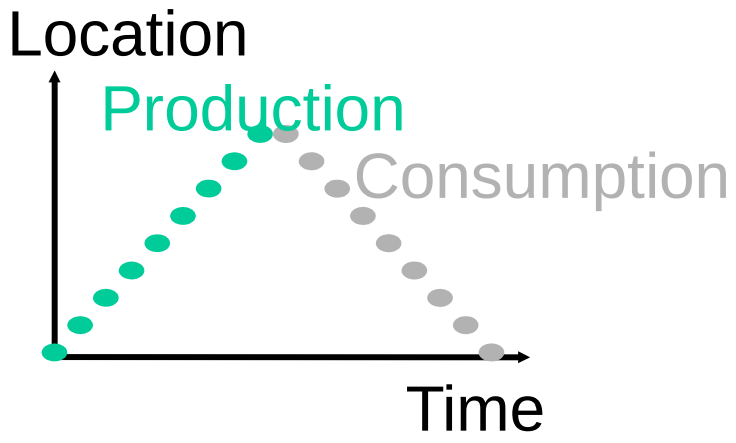
```
for (i=0; i<N; i++)
 A[i] = f(A[i-1]);
for (k=0; k<N; k++)
 B[k] = g(in[k]);
for (j=0; j<N; j++)
 C[j] = h(B[j],A[N]);
```

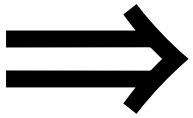
Loop Merge

```
for (i=0; i<N; i++)
 A[i] = f(A[i-1]);
for (j=0; j<N; j++)
 B[j] = g(in[j]);
 C[j] = h(B[j],A[N]);
```

Loop Reverse

for I = exp_1 to exp_2
    A(I)

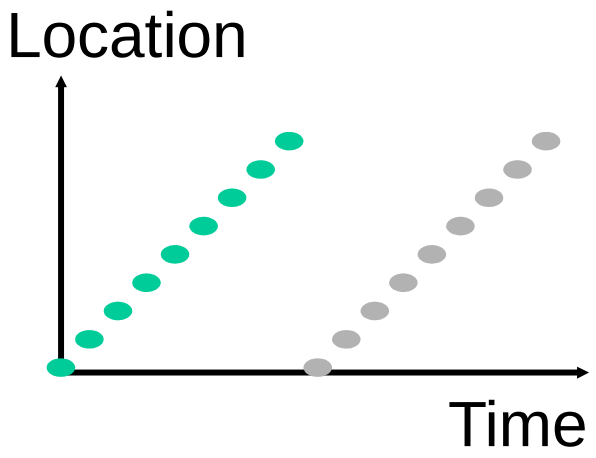




for I = exp_2 downto exp_1
    A(I)

Or written differently:

for I = exp_1 to exp_2
    A(exp_2-(I-exp_1))



Loop Reverse: Satisfy dependencies

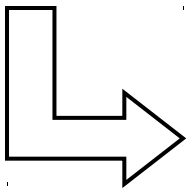
```
for (i=0; i<N; i++)
   A[i] = f(A[i-1]);
```

No loop-carried dependencies allowed ! Can not reverse the loop

```
A[0] = …
for (i=1; i<=N; i++)
   A[i]= A[i-1]+ f(…);

… = A[N] …
```

Enabler: data-flow transformations; exploit associativity of + operator



```
A[N] = …
for (i=N-1; i>=0; i--)
   A[i]= A[i+1]+ f(…);

… = A[0] …
```

Loop Reverse: Example as enabler

```
for (i=0; i<=N; i++)
   B[i] = f(A[i]);
for (i=0; i<=N; i++)
   C[i] = g(B[N-i]);
```

N–i > i ⇒ merging not possible

Loop Reverse ⇒

```
for (i=0; i<=N; i++)
   B[i] = f(A[i]);
for (i=0; i<=N; i++)
   C[N-i] = g(B[N-(N-i)]);
```

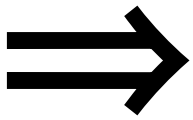
N-(N-i) = i ⇒ merging possible

Loop Merge ⇒

```
for (i=0; i<=N; i++)
    B[i] = f(A[i]);
    C[N-i] = g(B[i]);
```

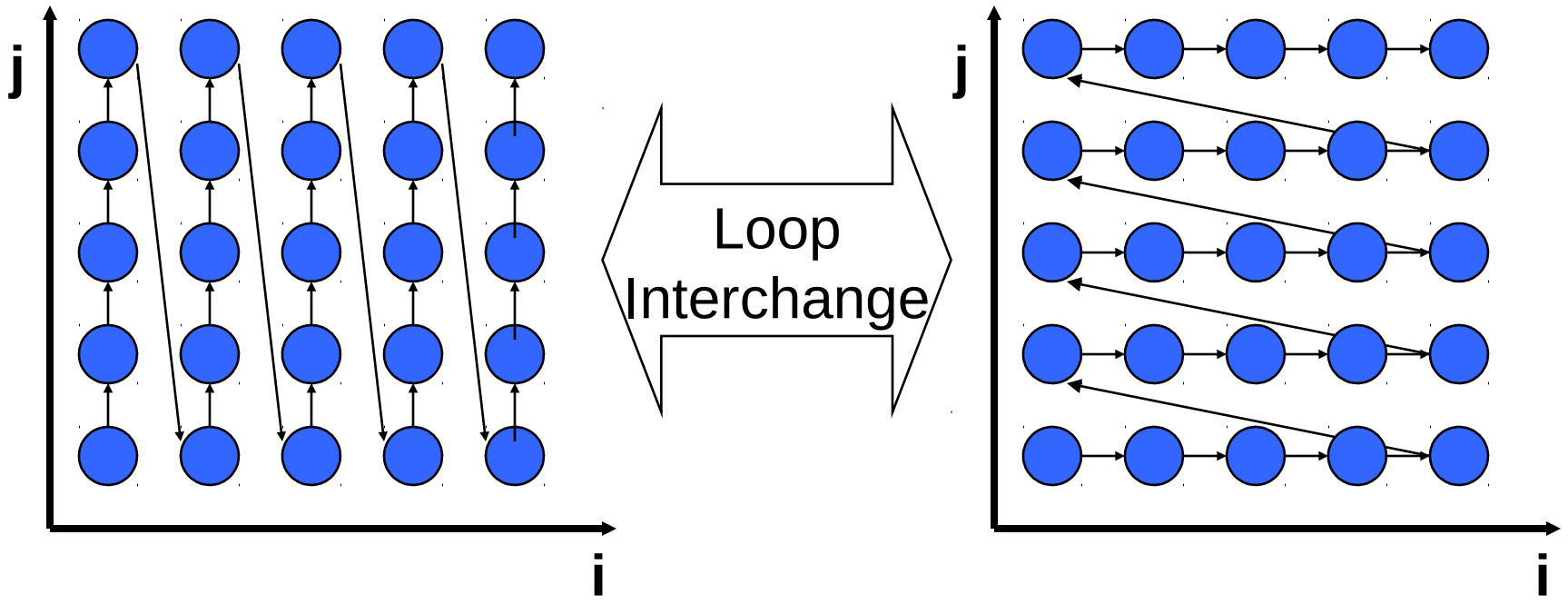
Loop Interchange

for I_1 = exp_1 to exp_2
    for I_2 = exp_3 to exp_4
        $A(I_1, I_2)$



for I_2 = exp_3 to exp_4
    for I_1 = exp_1 to exp_2
        $A(I_1, I_2)$

Loop Interchange: index traversal




```
for(i=0; i<W; i++)
  for(j=0; j<H; j++)
    A[i][j] = …;
```

```
for(j=0; j<H; j++)
  for(i=0; i<W; i++)
    A[i][j] = …;
```