



# Delivering Parallel Programmability to the Masses via the Intel MIC Ecosystem: A Case Study

**Kaixi Hou**, Hao Wang, and Wu-chun Feng











*Department of Computer Science, Virginia Tech*

# Intel Xeon Phi in HPC





\* Released in June 2014

- In the Top500 list\* of supercomputers ...
  - 27% of accelerator-based systems use Intel Xeon Phi (17/62)
  - Top 10:

- 1 Tianhe-2 
- 2 Titan 
- 3 Sequoia 
- 4 K computer 
- 5 Mira 
- 6 Piz Daint 
- 7 Stampede 
- 8 JUQUEEN 
- 9 Vulcan 
- 10 Cray XC30 

Equipped with Xeon Phi

	Name	Rmax (petaflop/s)	Xeon Phi /Node
1	Tianhe-2 	33.86	3
7	Stampede 	5.17	2

# Intel Xeon vs. Intel Xeon Phi



- ▶ Less than 12 cores/socket
- ▶ Cores @ ~3GHz
- ▶ 256-bit vector units
- ▶ DDR3 80~100GB/s BW



- ▶ Up to 61 cores
- ▶ Cores @ ~1 GHz
- ▶ 512-bit vector units
- ▶ GDDR5 150GB/s BW

# Intel Xeon vs. Intel Xeon Phi



- x86 architecture and programming models

So, is it easy to write programs for the Xeon Phi?

Yes. it's easy to write and run programs on Phi.

... but optimizing performance on Phi is not easy!

# Architecture-Specific Solutions

- Transposition in FFT [Park13]
  - Reduce memory accesses via cross-lane intrinsics
- *Swendsen-Wang* multi-cluster algorithm [Wende13]
  - Maneuver the elements in registers via the data-reordering intrinsics
- Linpack benchmark [Heinecke13]
  - Reorganize the computation patterns and instructions via assembly code

If the optimizations are Xeon Phi-specific, the codes are not easy to write and portable.



# Performance, Programmability, and Portability

- It's more than performance ...
  - ... programmability and portability.
- *Solution*: directive-based optimizations + “simple” algorithmic changes.
  - @Cache
    - Blocking to create better memory access
  - @Vector Units
    - Pragmas + loop structure changes
  - @Many-cores
    - Pragmas
  - Find the optimal combination of parameters.

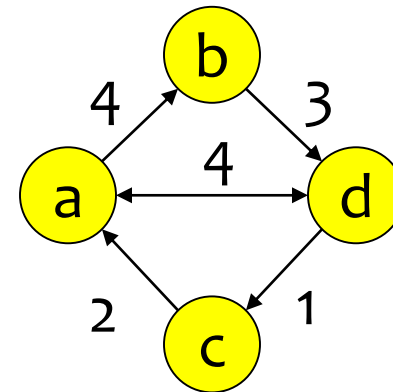
# Outline

- Introduction
  - Intel Xeon Phi
  - Architecture-Specific Solutions
- **Case Study : Floyd-Warshall Algorithm**
  - Algorithmic Overview
  - Optimizations for Xeon Phi
    - Cache Blocking
    - Vectorization via Data-Level Parallelism
    - Many Cores via Thread-Level Parallelism
  - Performance Evaluation on Xeon Phi
- Conclusion

# Case Study: Floyd-Warshall Algorithm

- *All-pairs shortest paths (APSP)* problem
- Algorithmic complexity:  $O(n^3)$
- Algorithmic Overview

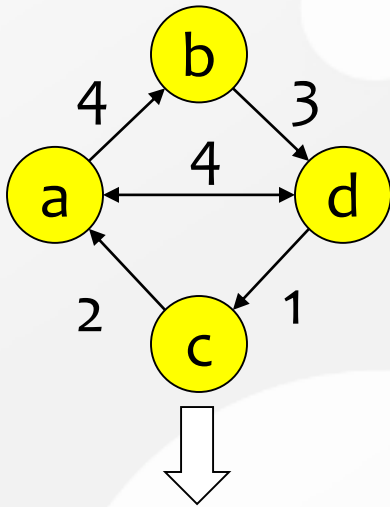
Keep an increasing subset of intermediate vertices for each iteration  
→ dynamic programming problem





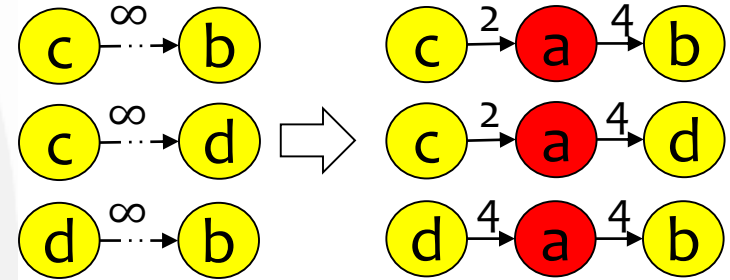
# A Quick Example

- $k$  means the newly added intermediate vertex in current iteration.



$k = 1$

	a	b	c	d
a	0	4	-	4
b	-	0	-	3
c	2	6	0	6
d	4	8	1	0



	a	b	c	d
a	0	4	-	4
b	-	0	-	3
c	2	-	0	-
d	4	-	1	0

$k = 2$

	a	b	c	d
a	0	4	-	4
b	-	0	-	3
c	2	6	0	6
d	4	8	1	0

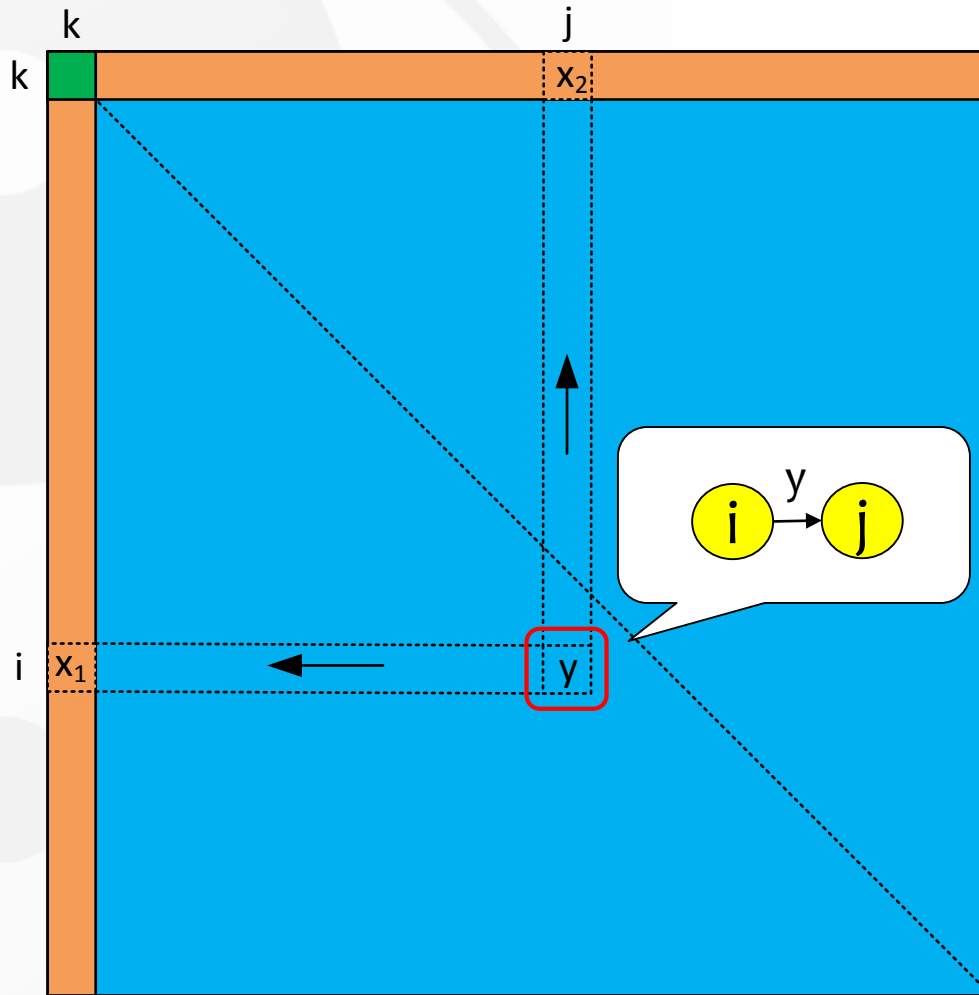
$k = 3$

	a	b	c	d
a	0	4	-	4
b	-	0	-	3
c	2	6	0	6
d	3	7	1	0

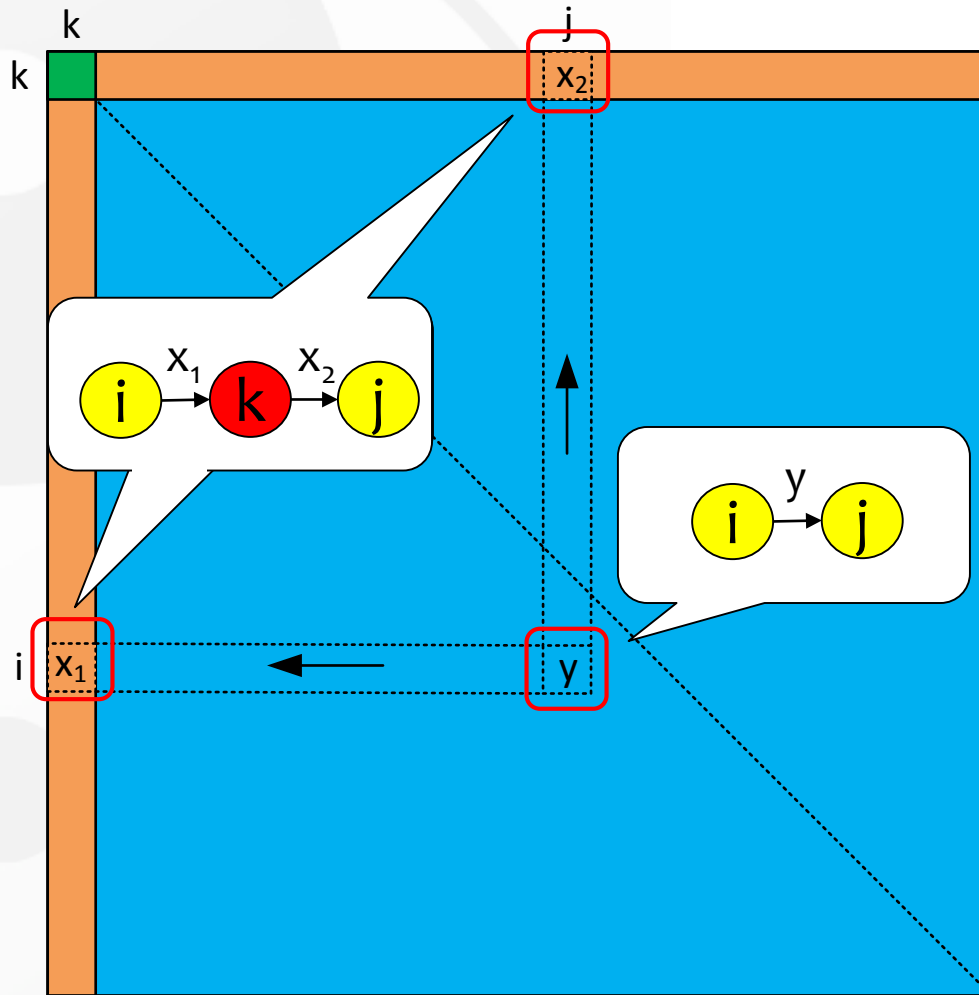
$k = 4$

	a	b	c	d
a	0	4	5	4
b	6	0	4	3
c	2	6	0	6
d	3	7	1	0

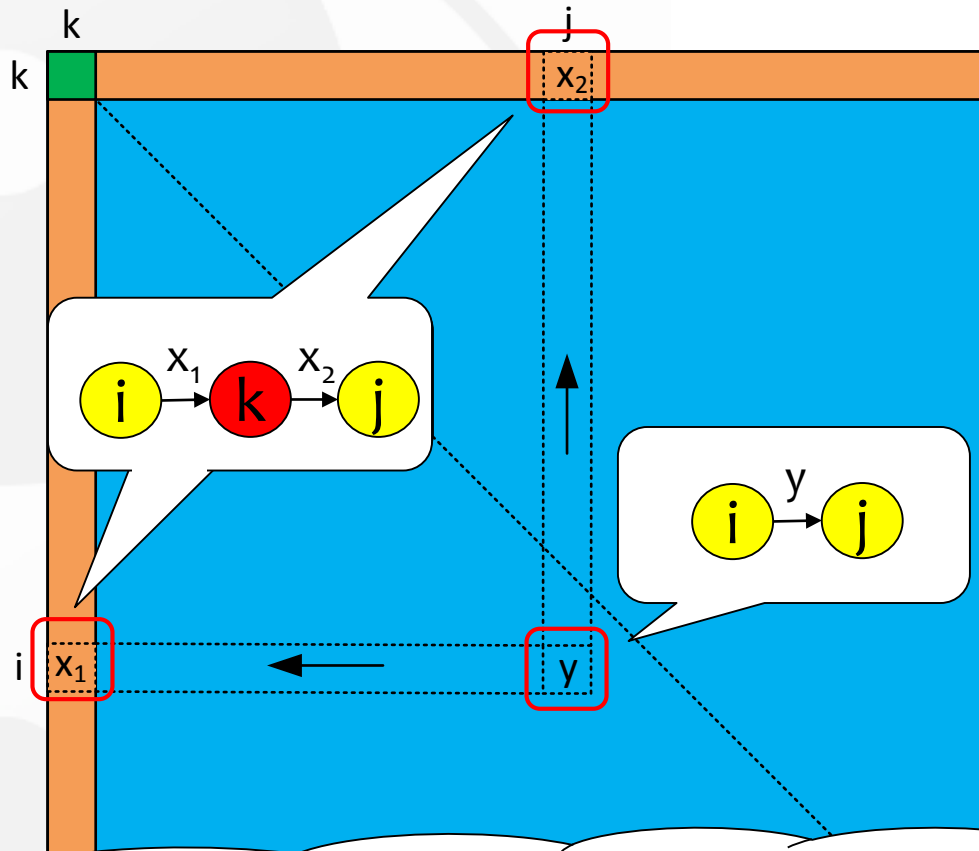
# Issue in Caching: Lack of Data Locality



# Issue in Caching: Lack of Data Locality

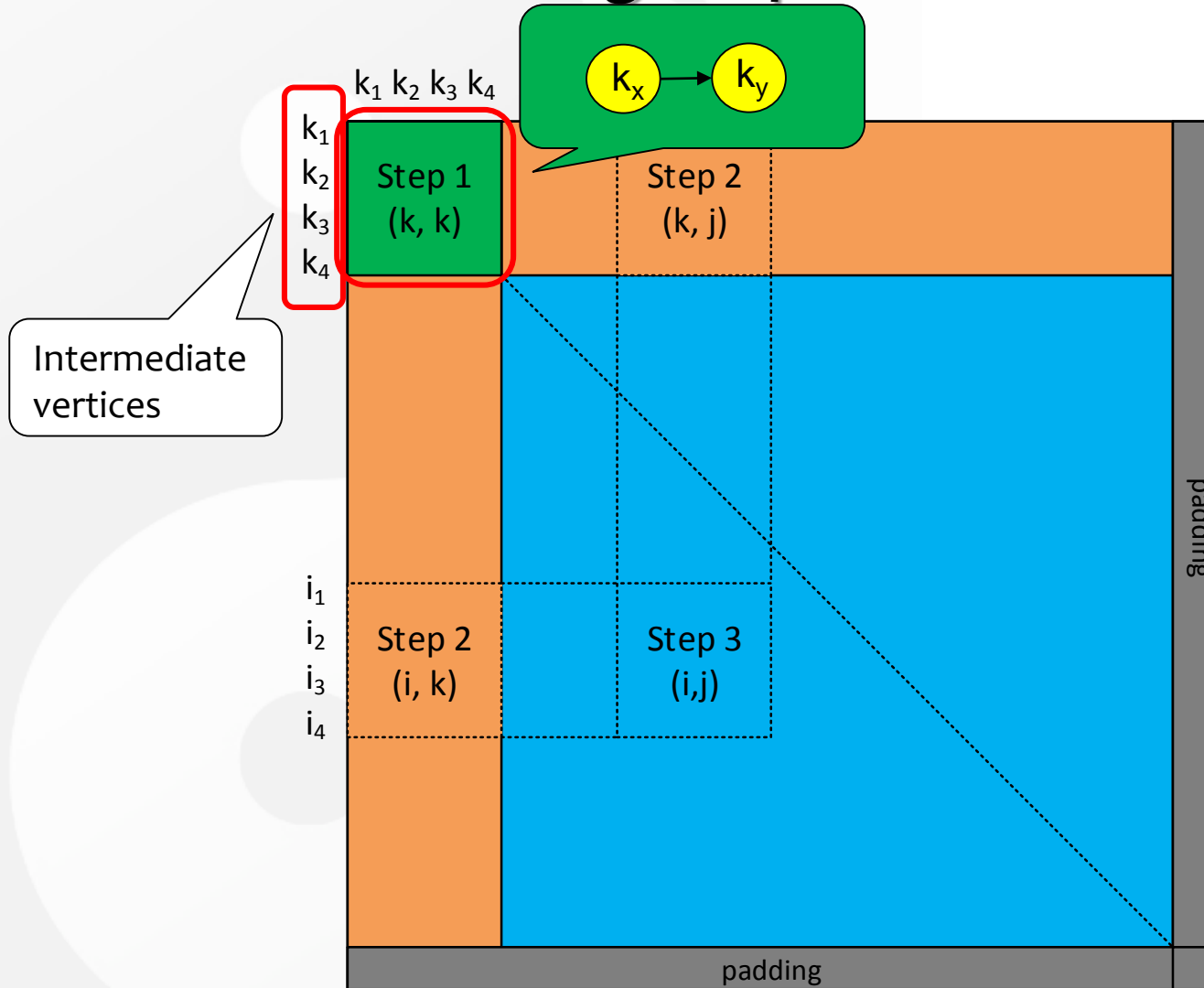


# Issue in Caching: Lack of Data Locality

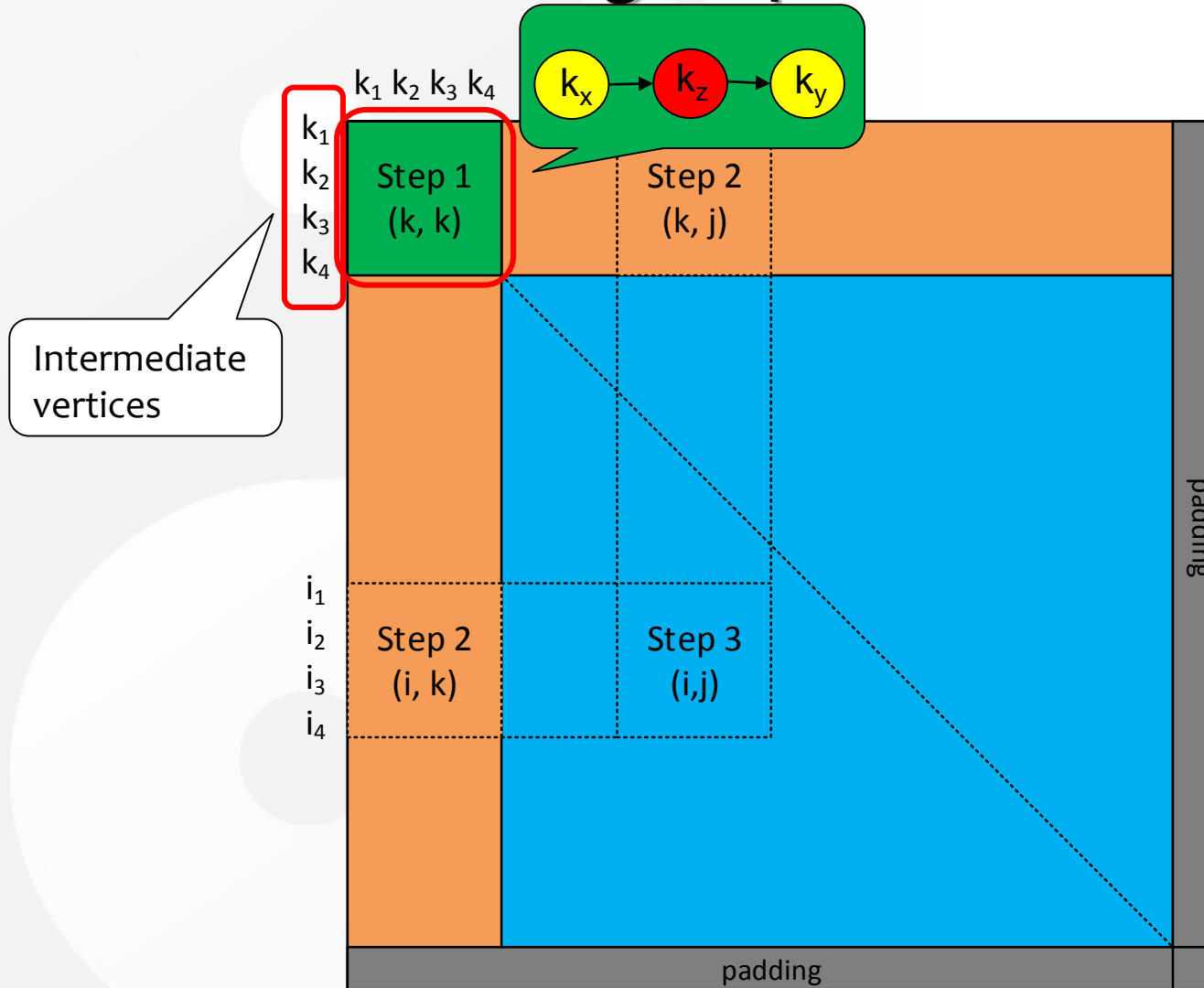


Default algorithm: data locality problem

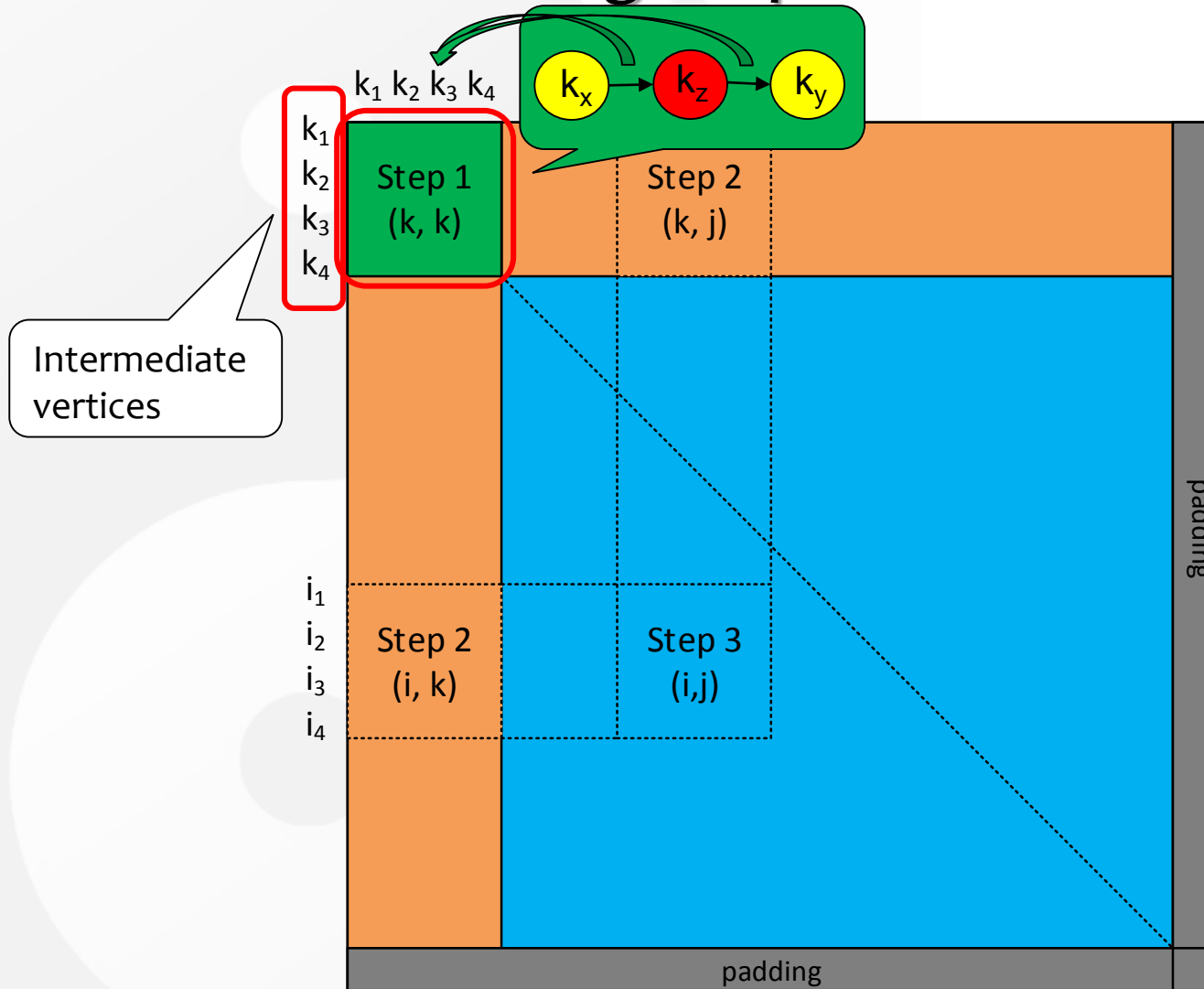
# Cache Blocking: Improve Data Reuse



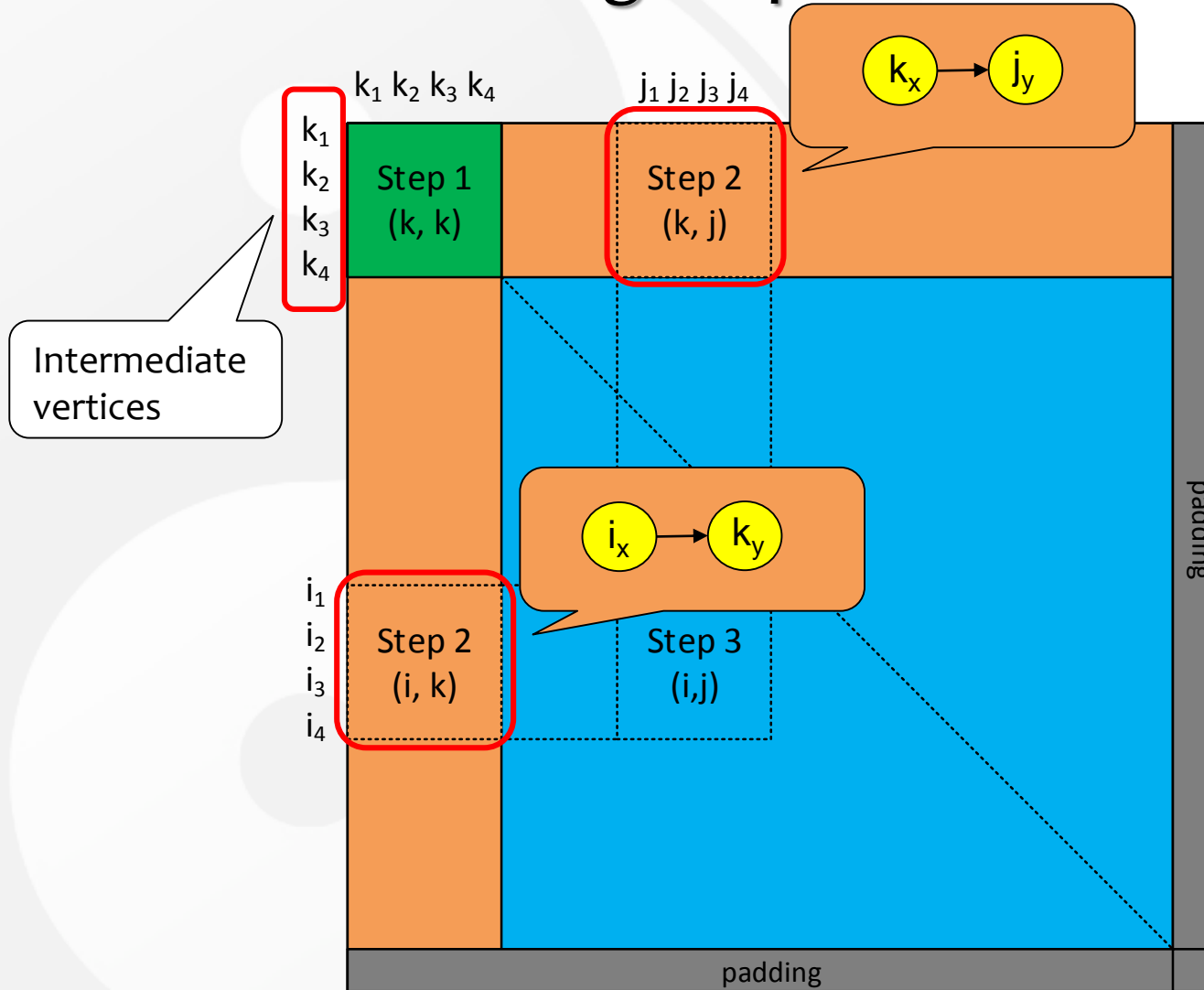
# Cache Blocking: Improve Data Reuse



# Cache Blocking: Improve Data Reuse

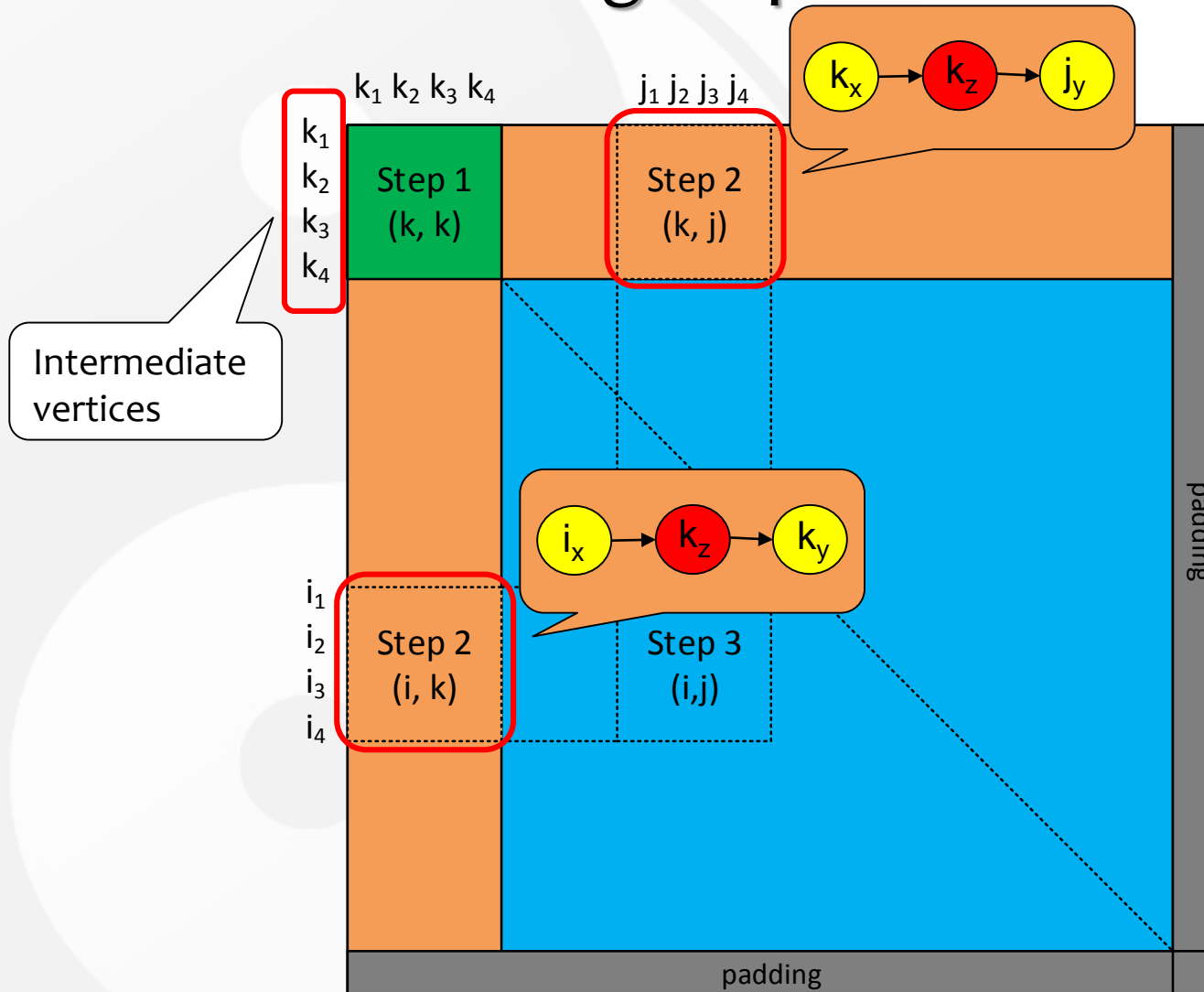


# Cache Blocking: Improve Data Reuse

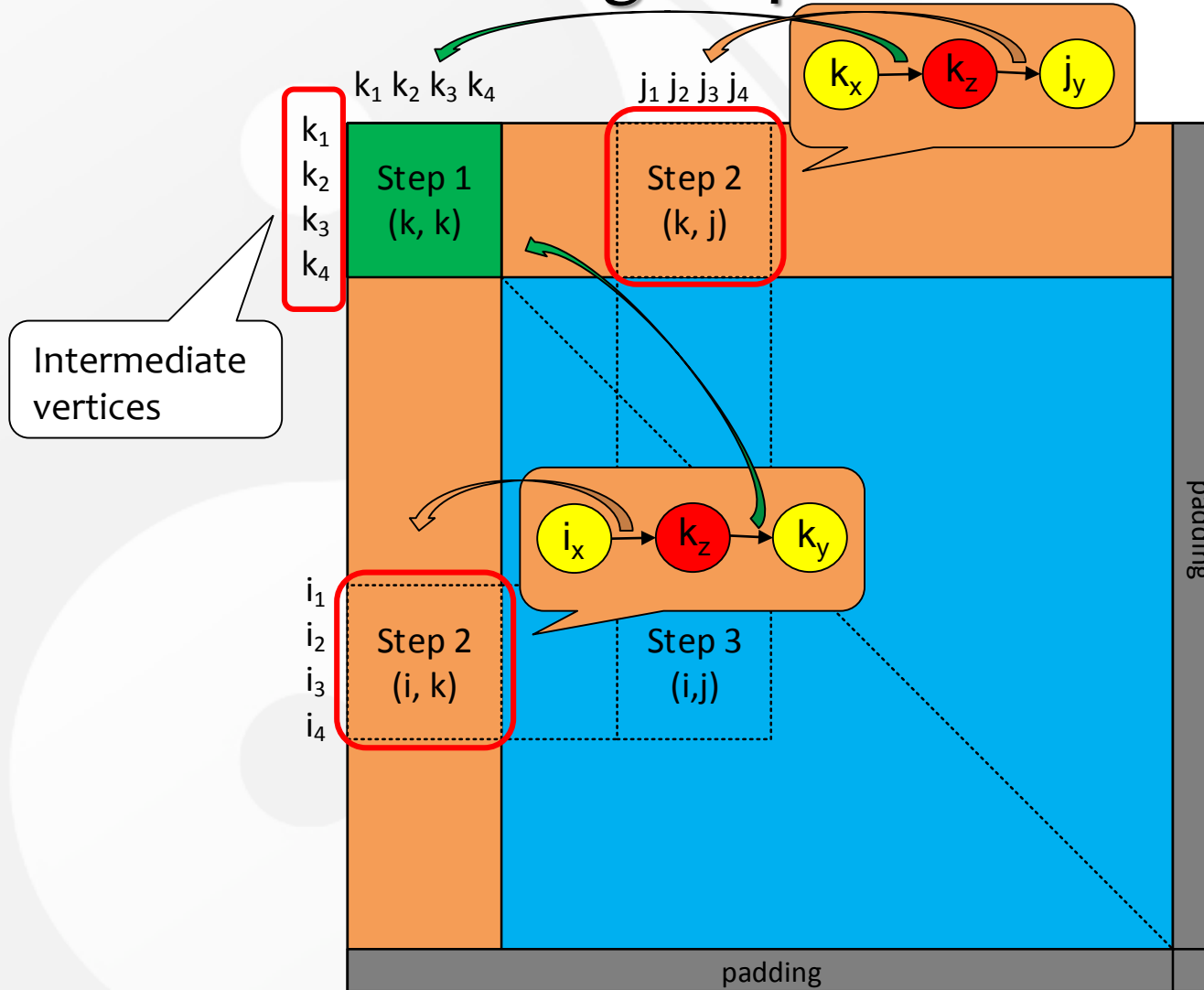




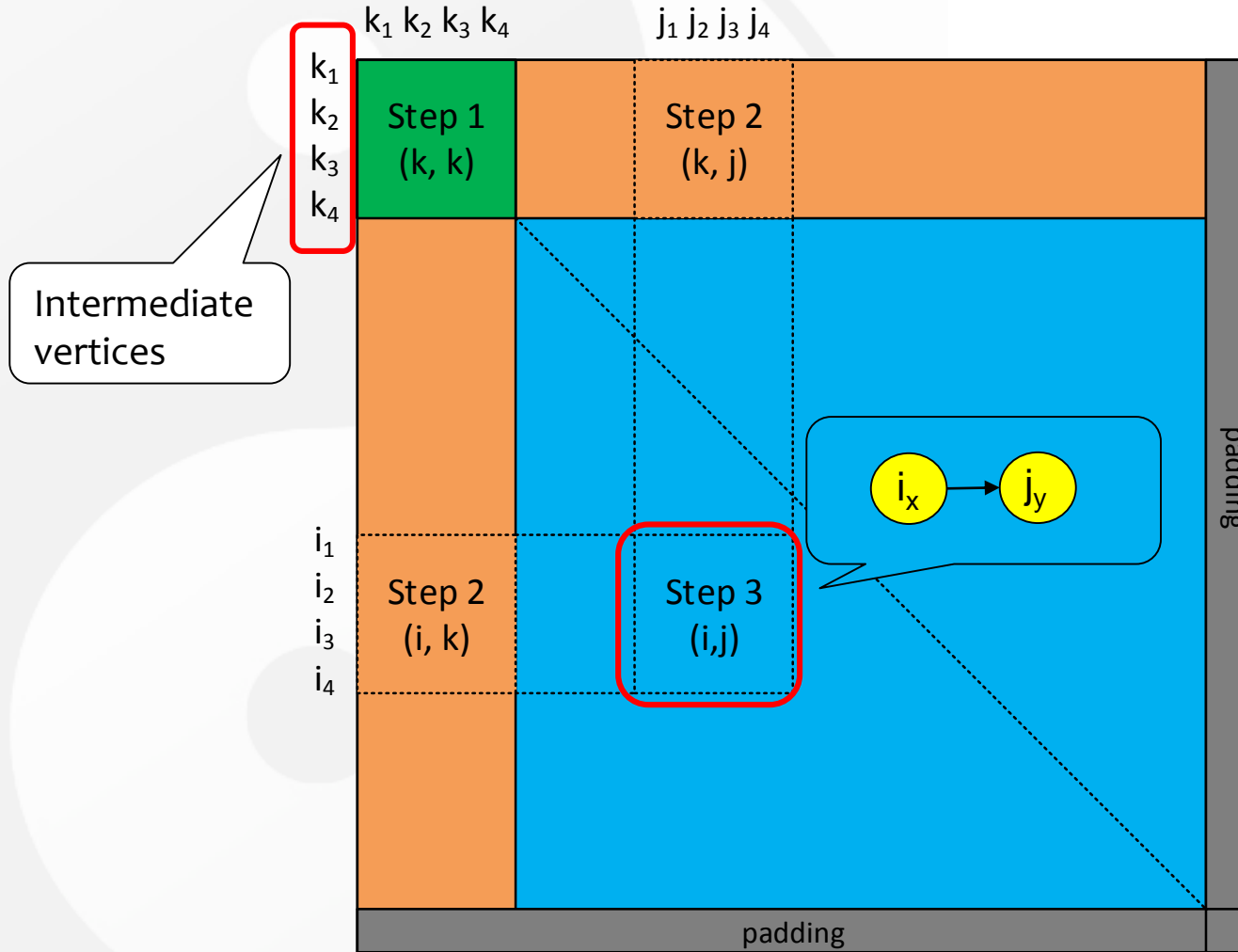
# Cache Blocking: Improve Data Reuse



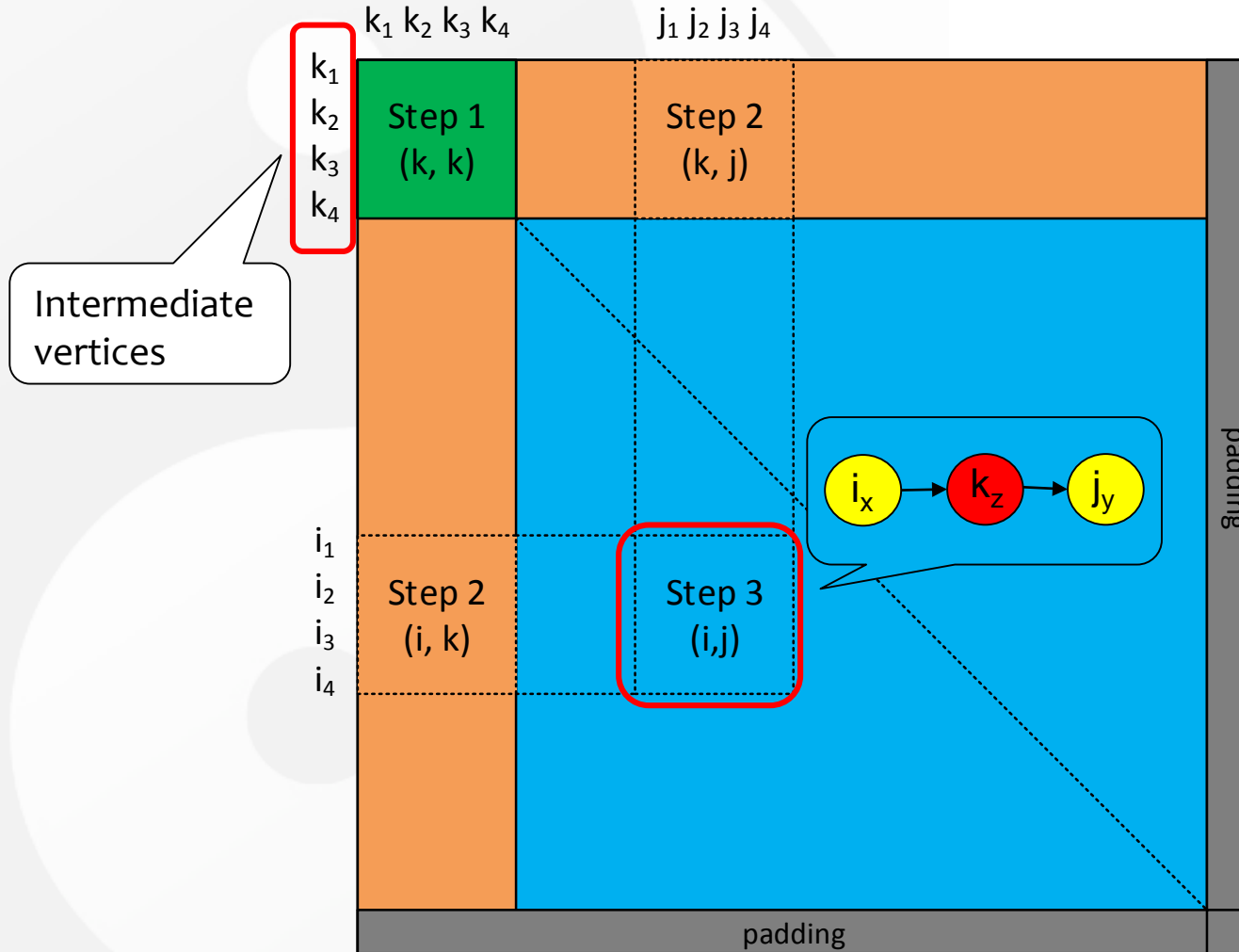
# Cache Blocking: Improve Data Reuse



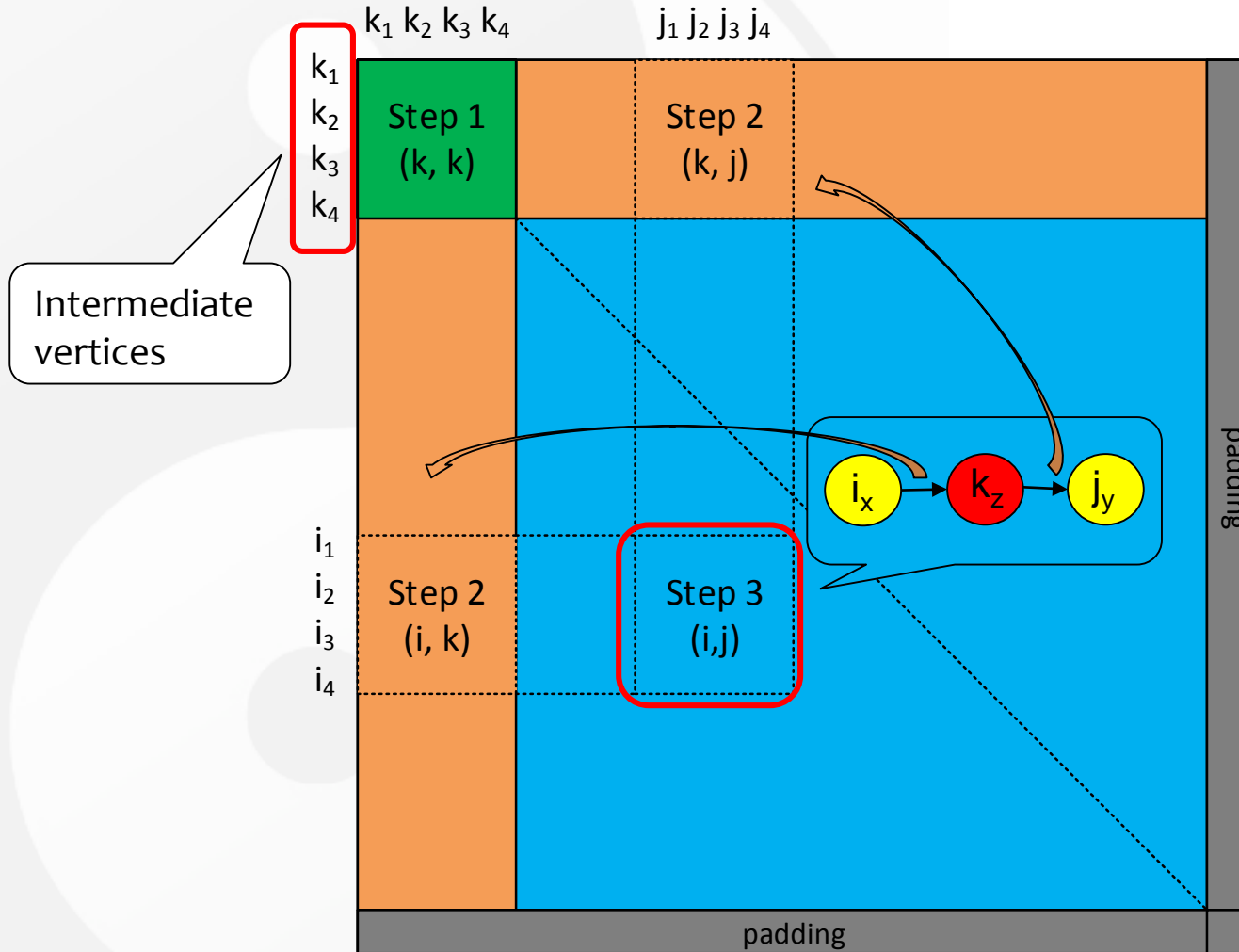
# Cache Blocking: Improve Data Reuse



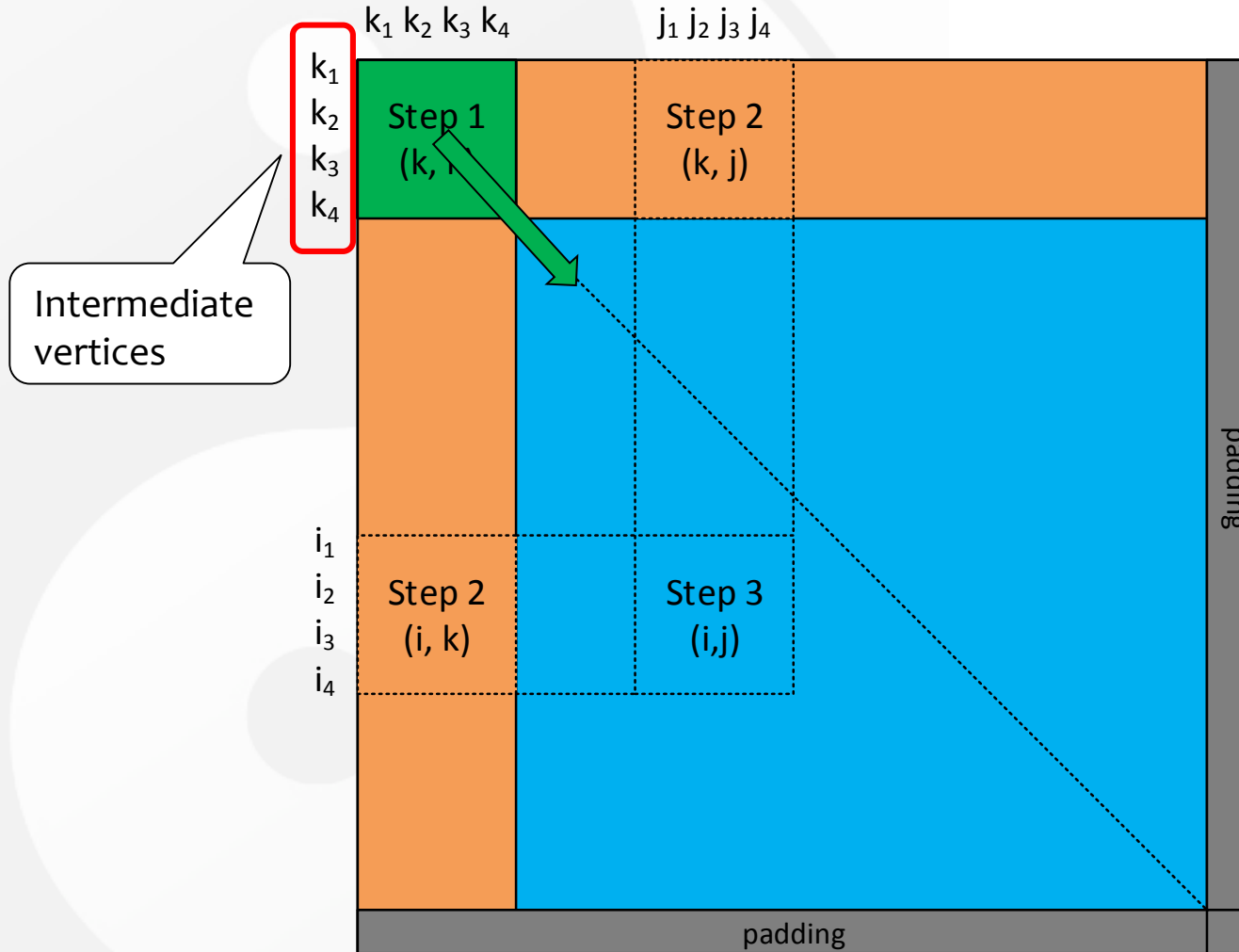
# Cache Blocking: Improve Data Reuse



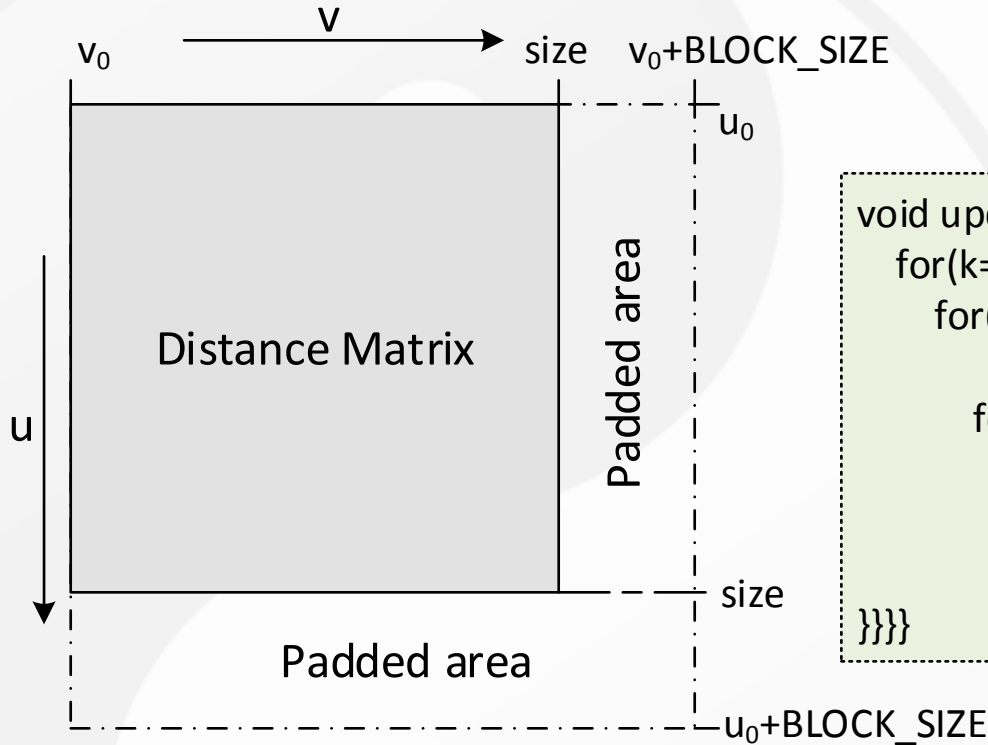
# Cache Blocking: Improve Data Reuse



# Cache Blocking: Improve Data Reuse



# Vectorization: Data-Level Parallelism



Version 1

```

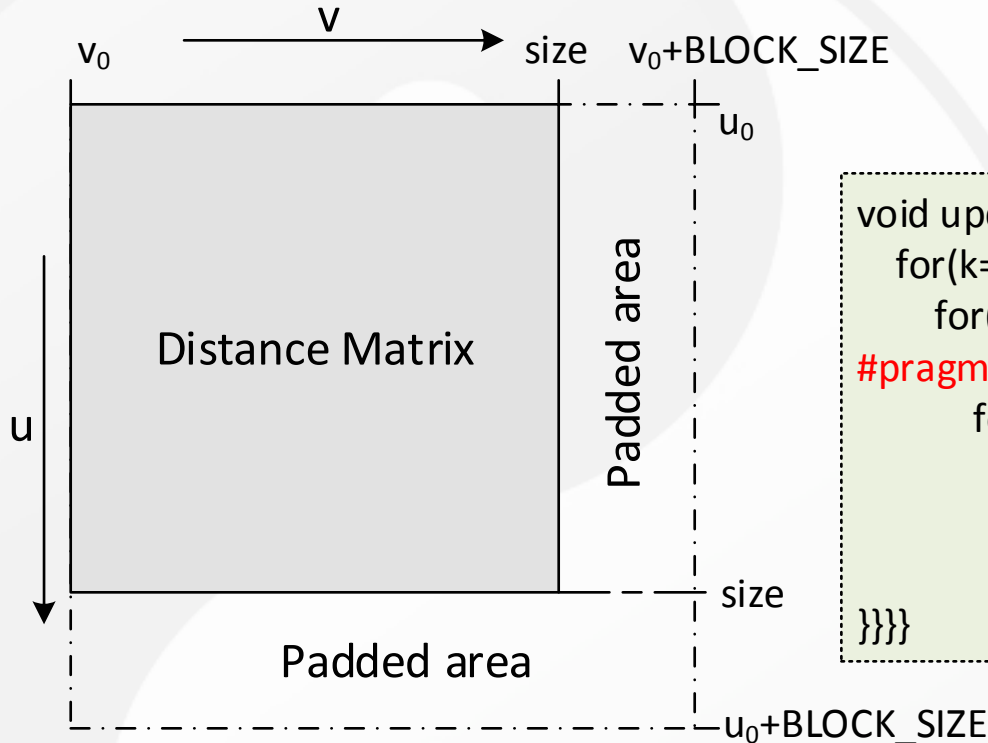
void update(int k0, int u0, int v0) {
  for(k=k0; k < MIN(k0+BLOCK_SIZE, size); k++) {
    for(u=u0; u < MIN(u0+BLOCK_SIZE, size); u++) {

      for(v=v0; v < MIN(v0+BLOCK_SIZE, size); v++) {
        if(dmat[u][v] > dmat[u][k] + dmat[k][v])
          dmat[u][v] = dmat[u][k] + dmat[k][v];
        pmat[u][v] = k;
      }
    }
  }
}
    
```

**Bottom-right block**

**Core computation**

# Vectorization: Data-Level Parallelism



Version 1

```
void update(int k0, int u0, int v0) {
    for(k=k0;k<MIN(k0+BLOCK_SIZE,size);k++) {
        for(u=u0;u<MIN(u0+BLOCK_SIZE,size);u++) {
            #pragma ivdep
                for(v=v0;v<MIN(v0+BLOCK_SIZE,size);v++) {
                    if(dmat[u][v]>dmat[u][k]+dmat[k][v])
                        dmat[u][v]=dmat[u][k]+dmat[k][v];
                    pmat[u][v]=k;
                }
            }
        }
    }
}
```

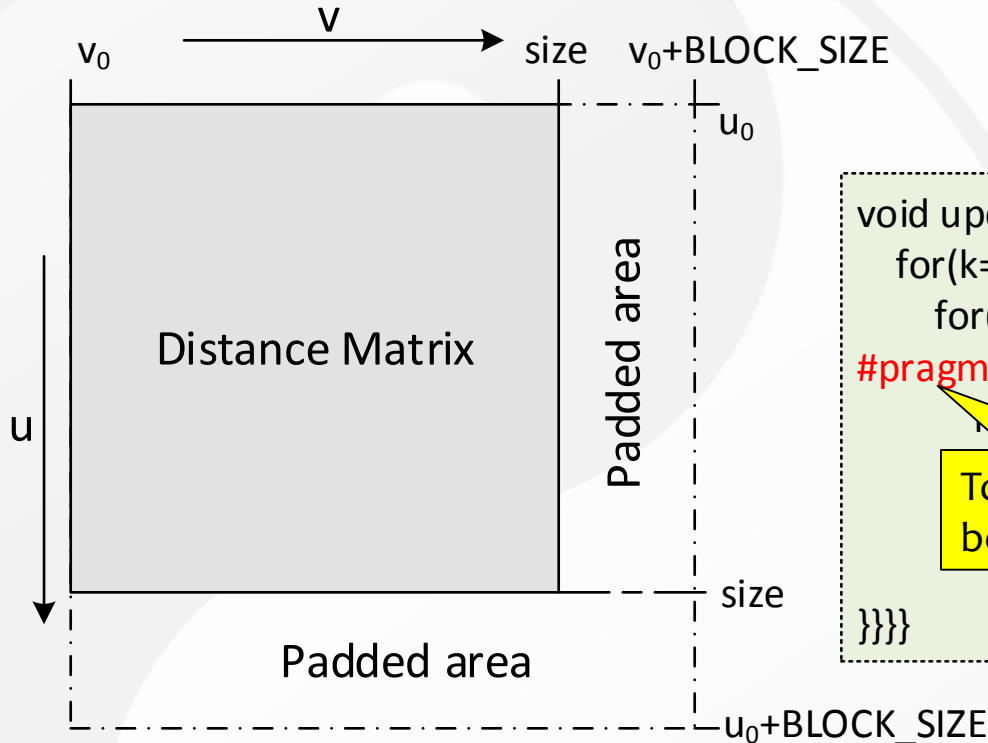
**Bottom-right block**

**Core computation**

- Pragmas to guide the compiler to vectorize the loop:
  - *#pragma vector always*: vectorize the loop regardless of the efficiency
  - *#pragma ivdep*: ignore vector dependencies



# Vectorization: Data-Level Parallelism



Version 1

```
void update(int k_0, int u_0, int v_0) {
    for(k=k_0;k<MIN(k_0+BLOCK_SIZE,size);k++) {
        for(u=u_0;u<MIN(u_0+BLOCK_SIZE,size);u++) {
            #pragma ivdep
            for(v=v_0;v<MIN(v_0+BLOCK_SIZE,size);v++) {
                u[k]+dmat[k][v]
                u[k]+dmat[k][v];
                pmat[u][v]=k,
            }
        }
    }
}
```

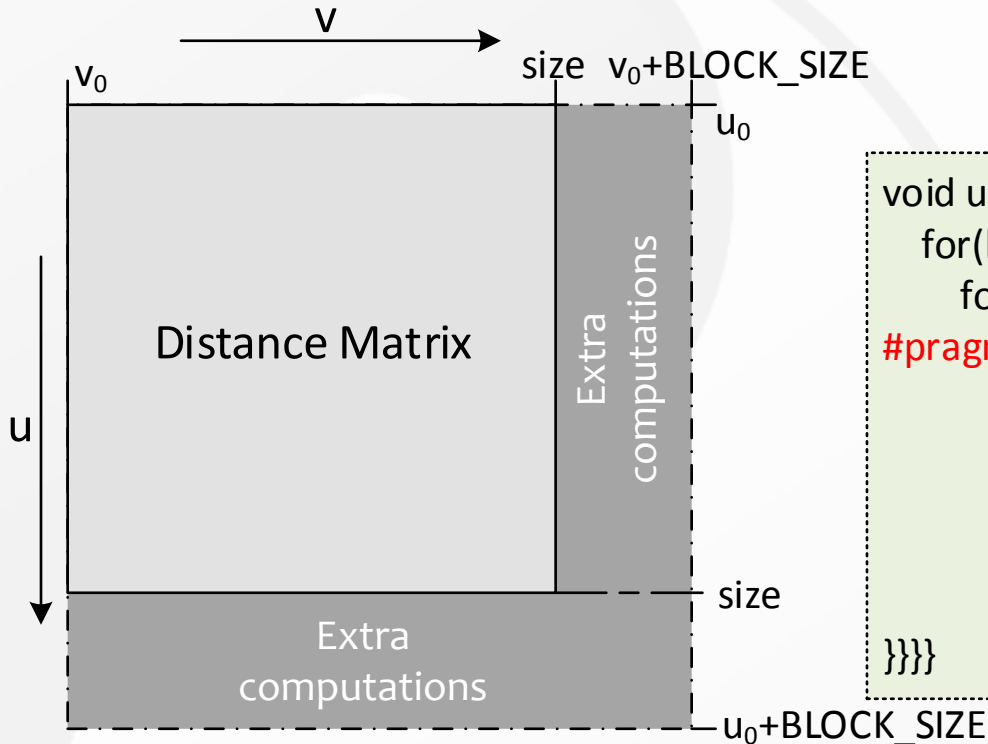
Top test could not be found.

Bottom-right block

Core computation

- Pragmas to guide the compiler to vectorize the loop:
  - *#pragma vector always*: vectorize the loop regardless of the efficiency
  - *#pragma ivdep*: ignore vector dependencies

# Vectorization: Data-Level Parallelism (Cont'd)



Version 3

```
void update(int k_0, int u_0, int v_0) {
  for(k=k_0; k<MIN(k_0+BLOCK_SIZE, size); k++){
    for(u=u_0; u<(u_0+BLOCK_SIZE); u++) {
      #pragma ivdep
      for(v=v_0; v<(v_0+BLOCK_SIZE); v++) {
        if(dmat[u][v]>dmat[u][k]+dmat[k][v])
          dmat[u][v]=dmat[u][k]+dmat[k][v];
        pmat[u][v]=k;
      }
    }
  }
}
```

## Bottom-right block

## SIMD-friendly codes

- Modify the boundary check conditions (u-loop & v-loop)
  - Extra computations but regular loop forms
- Keep boundary check condition (k-loop)
  - Where to fetch data

# Many Cores: Thread-Level Parallelism (TLP)

- OpenMP pragmas
  - A portable way to parallelize serial programs
  - Run-time specifications: thread number, thread affinity, etc.
- Utilize thread-level parallelism (TLP) in Xeon Phi
  - Apply OpenMP pragmas on loops of step 2 and step 3: most parallelism opportunities.

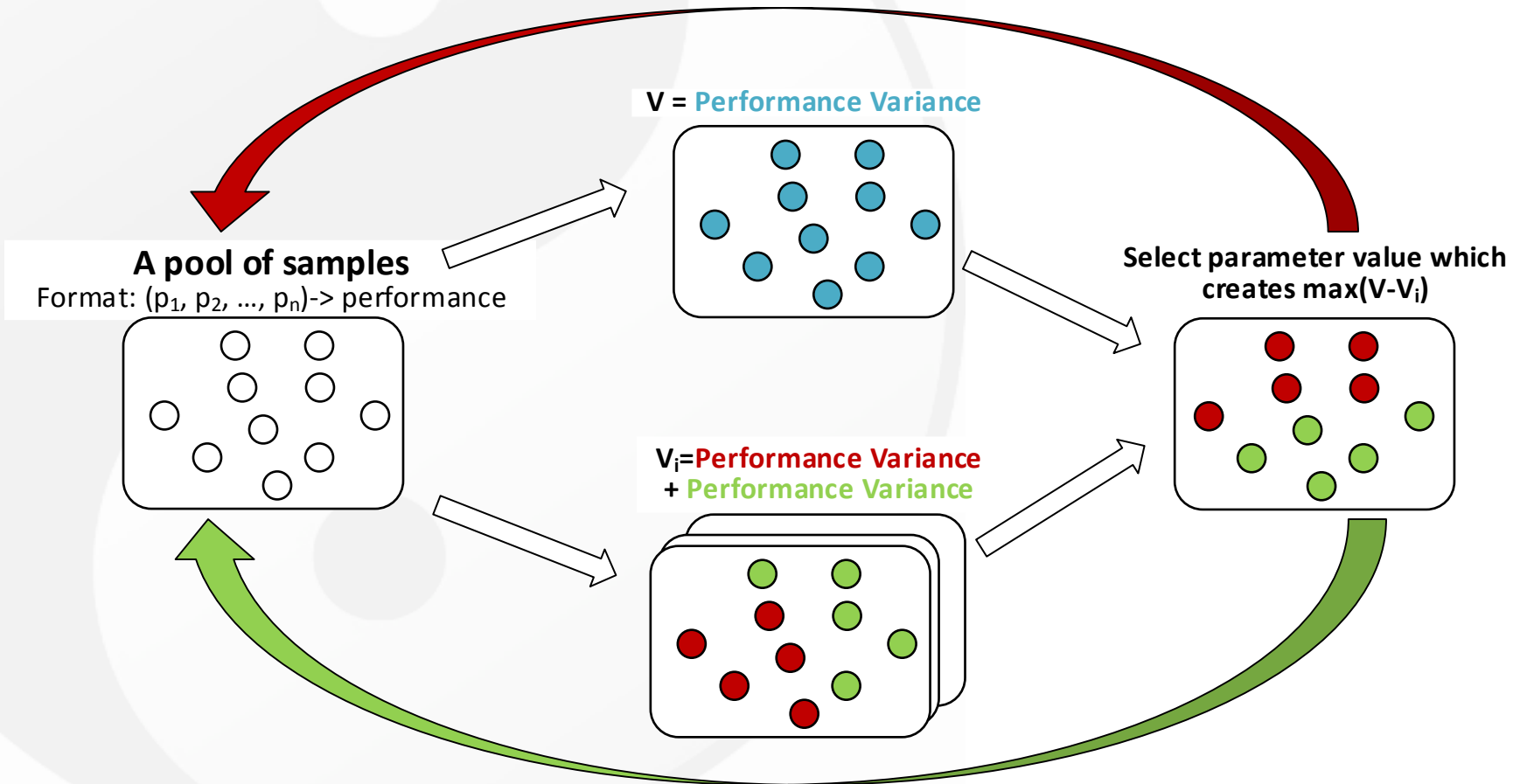
# Optimization *Challenges* in Thread-Level Parallelism

- Many configurable parameters
  - Ex: block size, thread number, runtime scheduling policy, etc.
- Difficulty in finding an appropriate *combination* of parameters
  - Inter-dependency between parameters
  - Huge search space

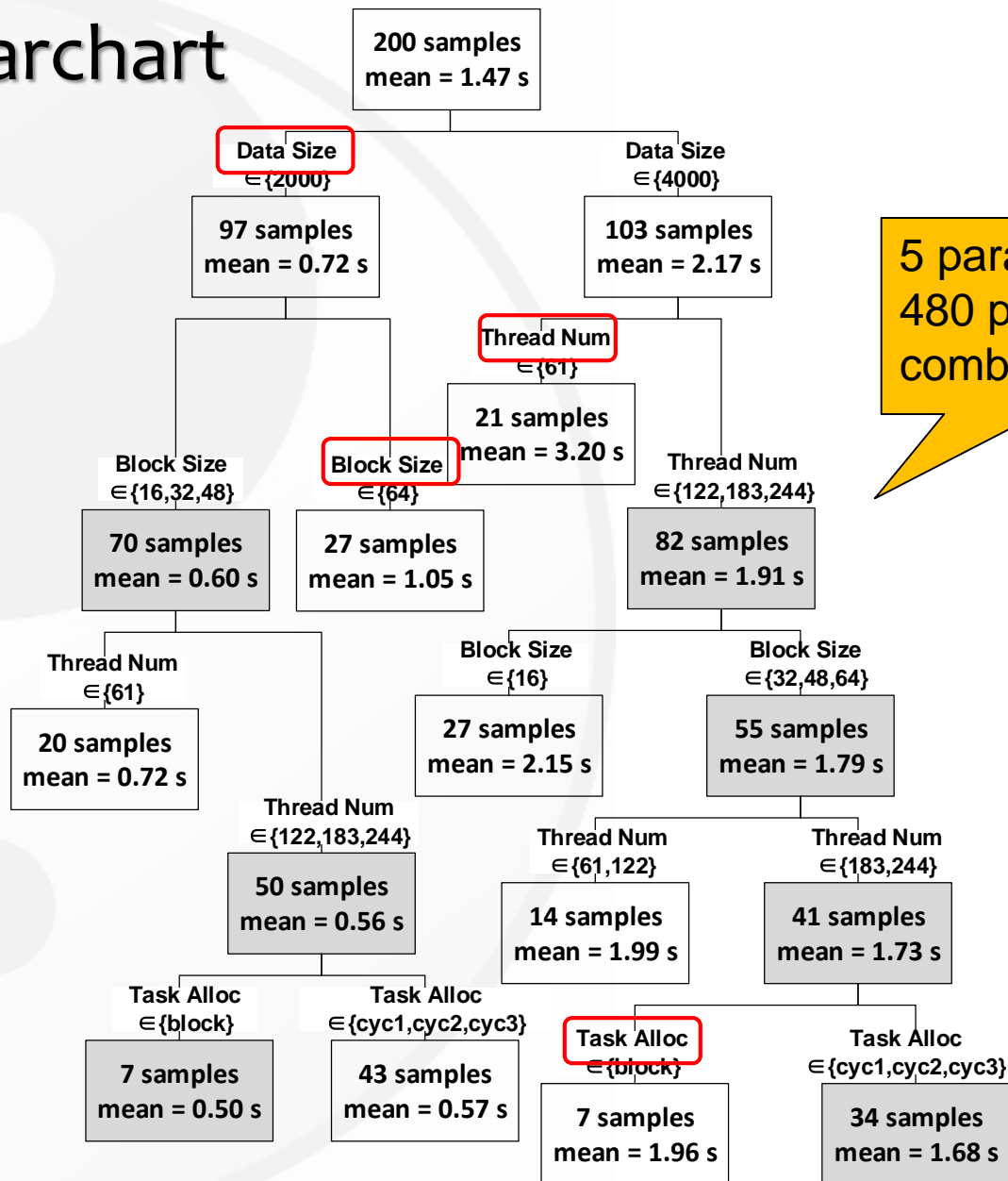
# Optimization Approach to Thread-Level Parallelism

- Starchart: Tree-based partitioning

[Jia13]

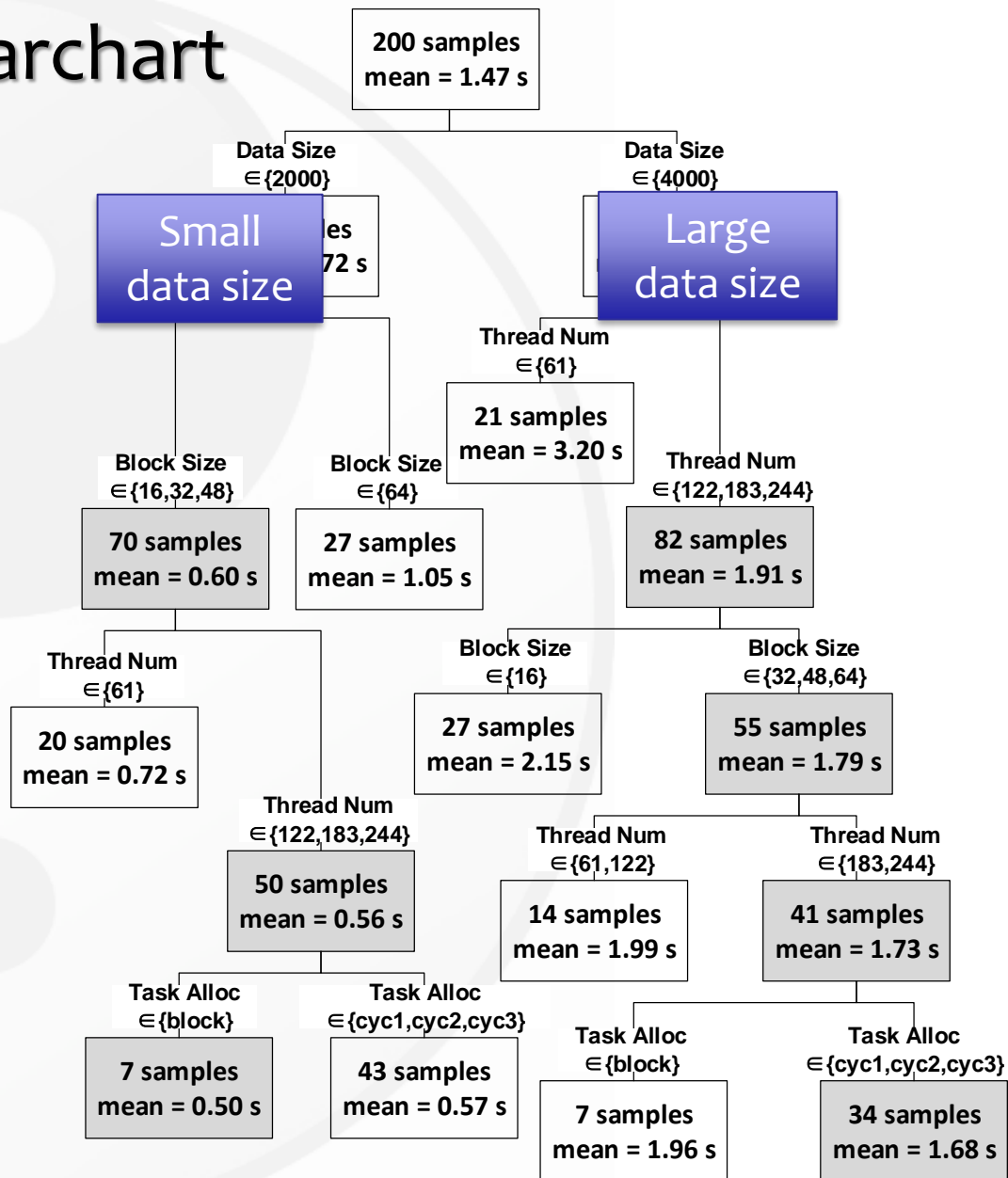


# Applying Starchart



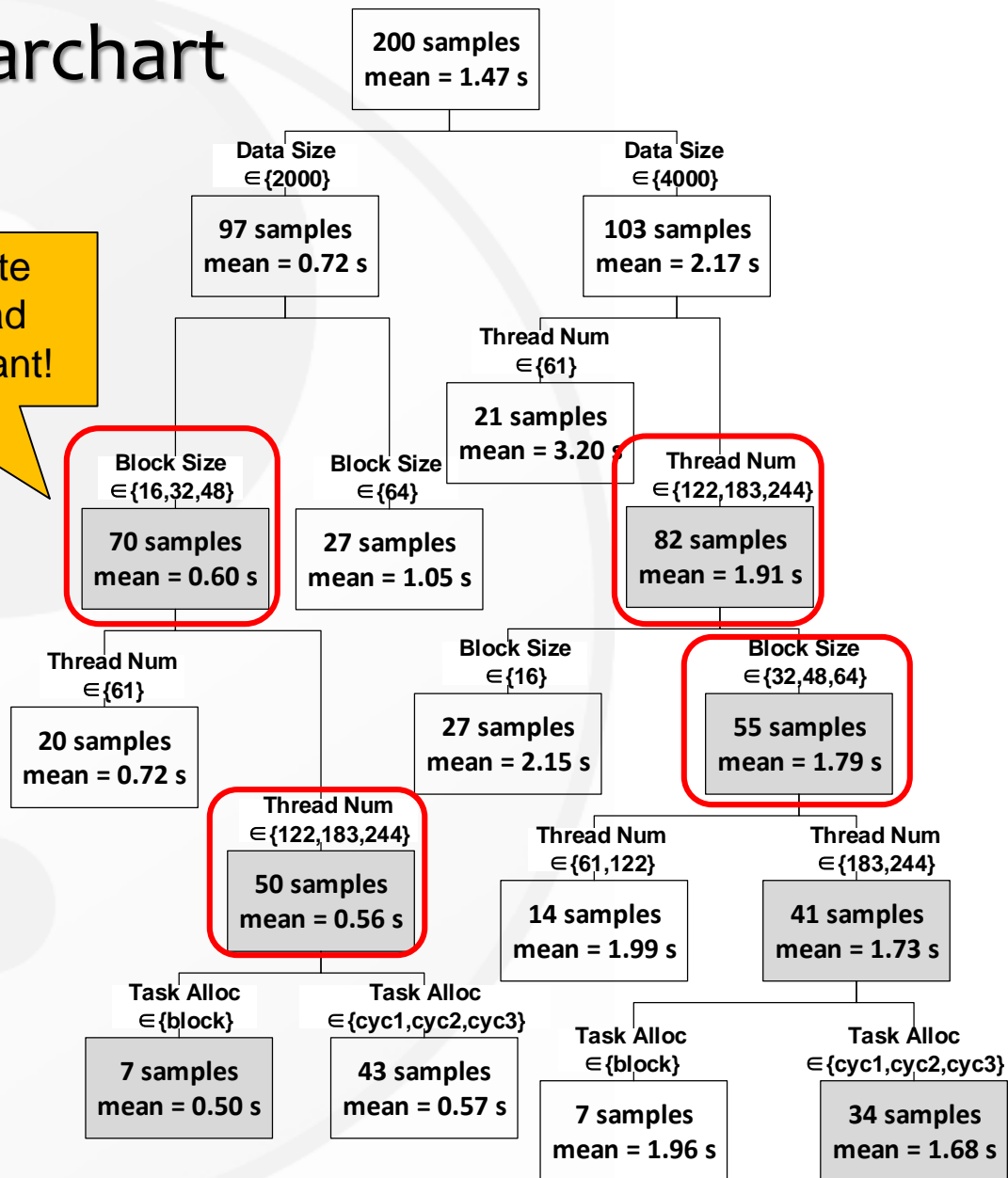
5 parameters:  
480 possible combinations

# Applying Starchart



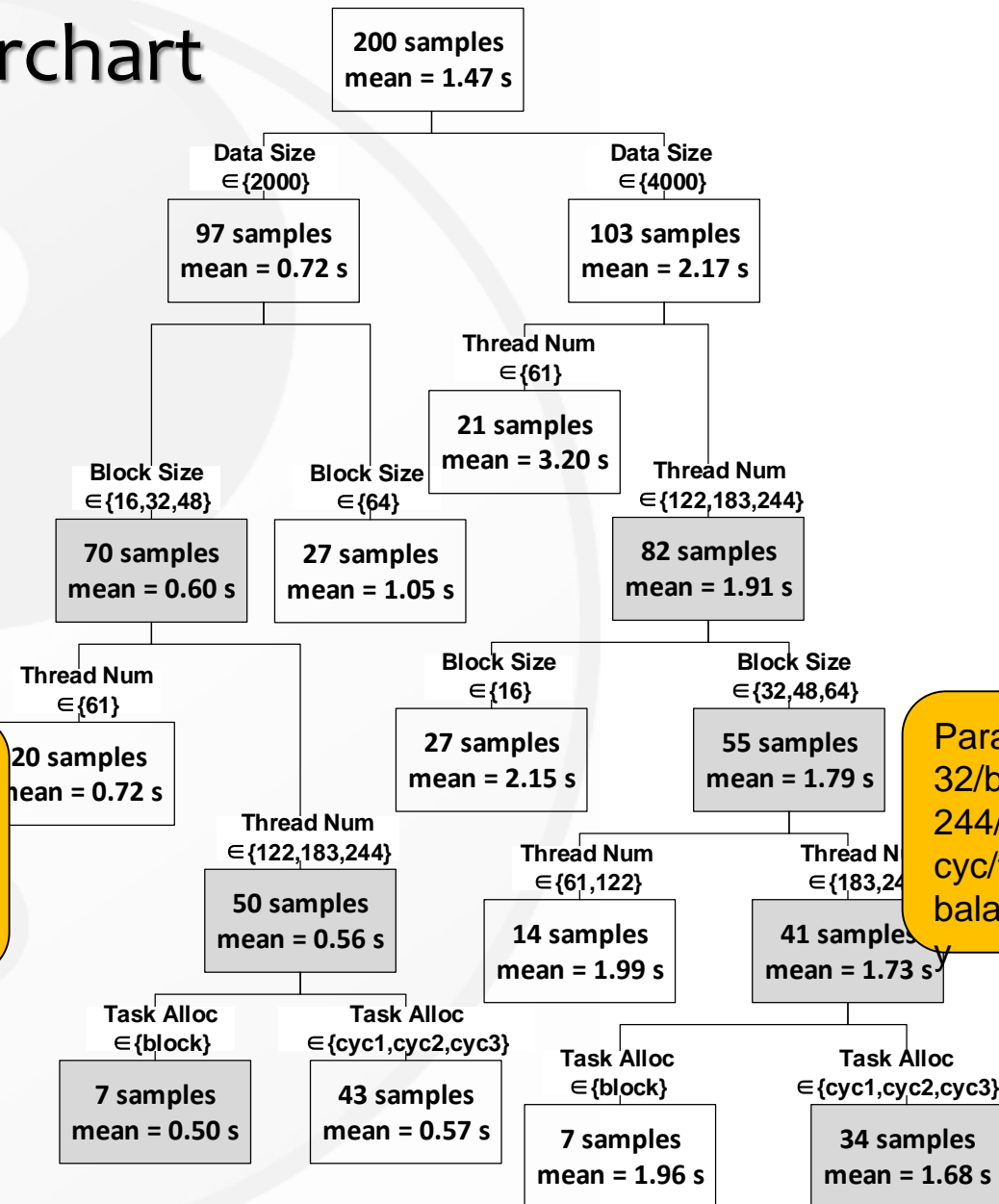
# Applying Starchart

Choosing appropriate block size and thread num is most important!





# Applying Starchart



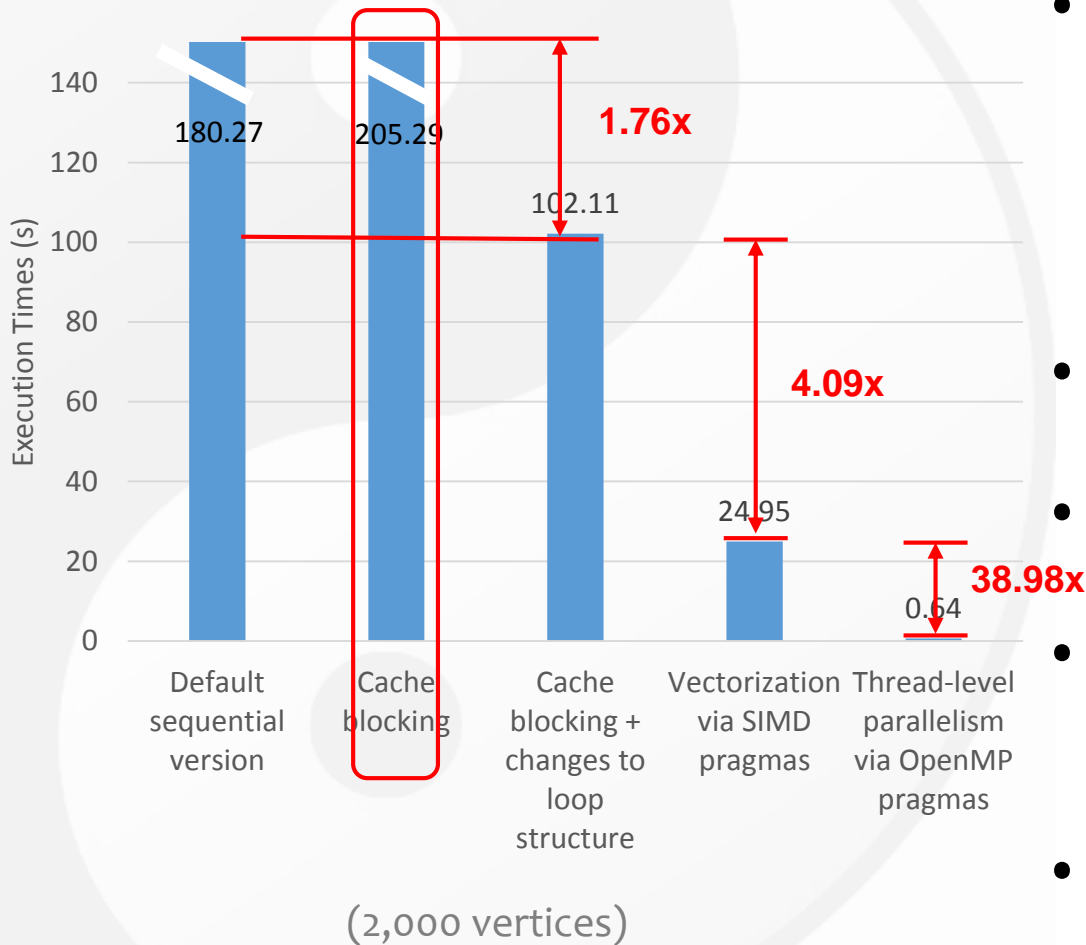
Parameters (small):  
 32 /blockSize,  
 244/threadNum,  
 block/taskAlloc,  
 balanced/threadAffinity

Parameters (large):  
 32/blockSize,  
 244/threadNum,  
 cyc/taskAlloc,  
 balanced/threadAffinity

# Outline

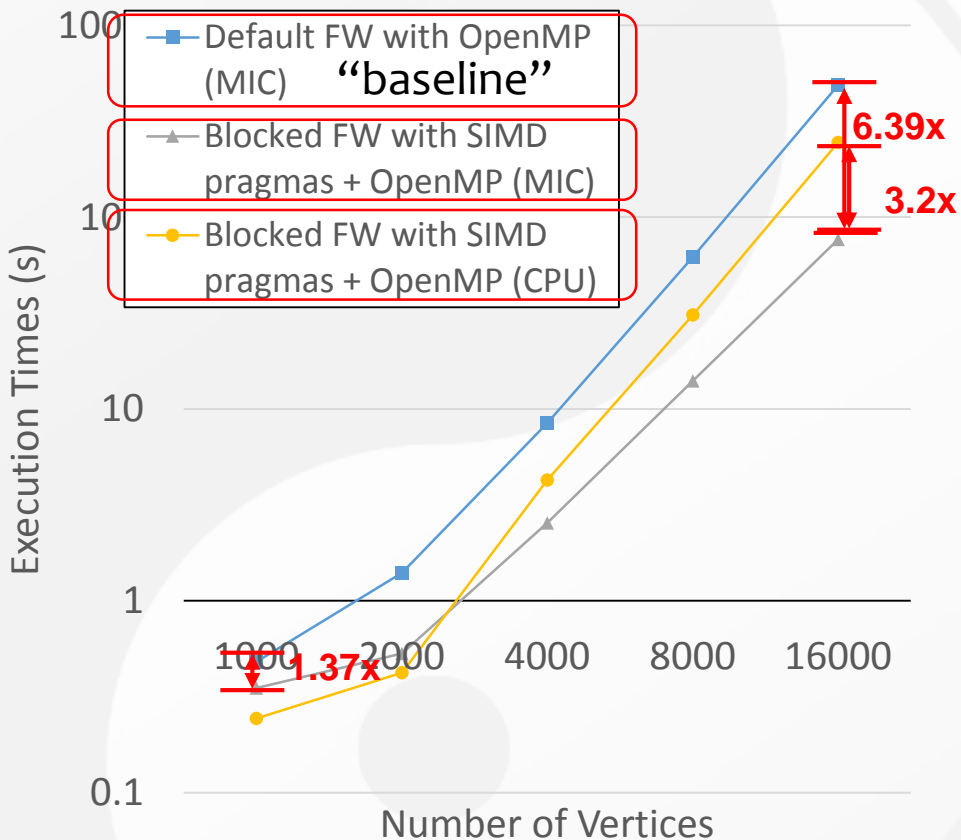
- Introduction
  - Intel Xeon Phi
  - Architecture-Specific Solutions
- **Case Study : Floyd-Warshall Algorithm**
  - Algorithmic Overview
  - Optimizations for Xeon Phi
    - Cache Blocking
    - Vectorization via Data-Level Parallelism
    - Many Cores via Thread-Level Parallelism
  - Performance Evaluation for Xeon Phi
- Conclusion

# Performance Evaluation : Step-by-Step



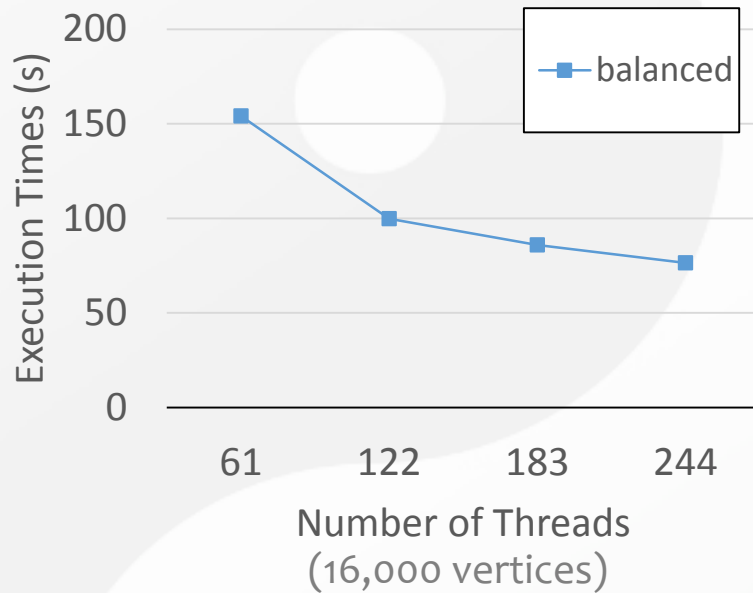
- **Cache blocking :**  
14% performance loss
  - Redundant computations induced in step 2 and step 3.
  - Boundary check conditions in the loop structures
- **Cache blocking with changes to loop structure : 1.76x**
- **Vectorization via SIMD pragmas : 4.09x**
- **Thread-level parallelism via OpenMP pragmas : 38.98x**
- Overall: 281.67x

# Performance Evaluation : Scalability

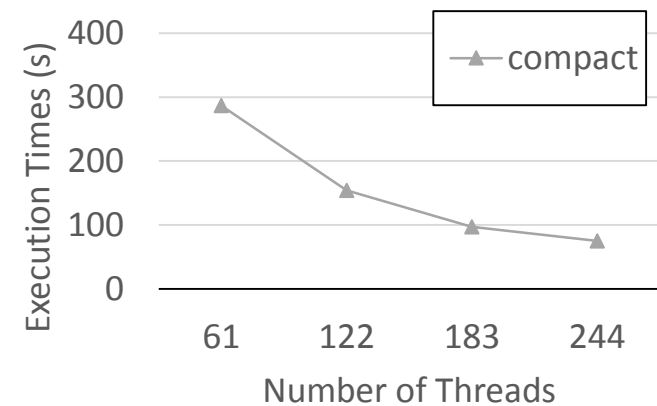
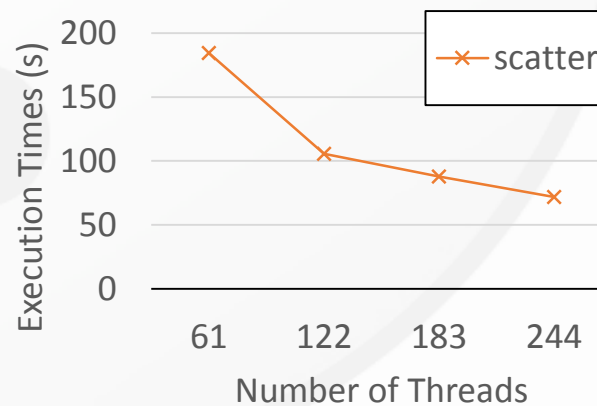


- Baseline: OpenMP version of default algorithm
- Optimized (MIC) vs. Baseline: up to 6.39x
- Optimized (MIC) vs. Optimized (CPU): up to 3.2x
  - Peak performance ratio of MIC and CPU: 3.23x (2148 Gflops and 665.6 Gflops)

# Performance Evaluation : Strong Scaling



- Balanced thread affinity:
  - 2x from 1 thread/core to 4 threads/core
- Other affinities:
  - Scatter: 2.6x
  - Compact: 3.8x



# Conclusion

- CPU programs can be recompiled and directly run on Intel Xeon Phi, but achieving optimized performance requires a considerable effort.
  - Considerations: Performance, programmability, and portability
- We use directive-based optimizations and certain algorithmic changes to achieve significant performance gains for the Floyd-Warshall algorithm as a case study.
  - 6.4x speedup over a default OpenMP version of Floyd-Warshall on Xeon Phi.
  - 3.2x speedup over a 12-core multicore CPU (Sandy Bridge).



Thanks! Questions?