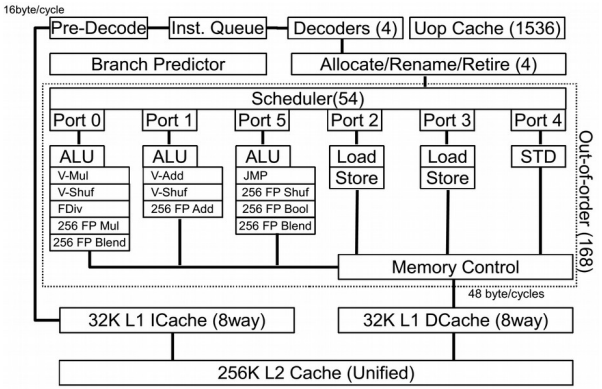
Advanced Optimization Techniques

ICTP Trieste 2014

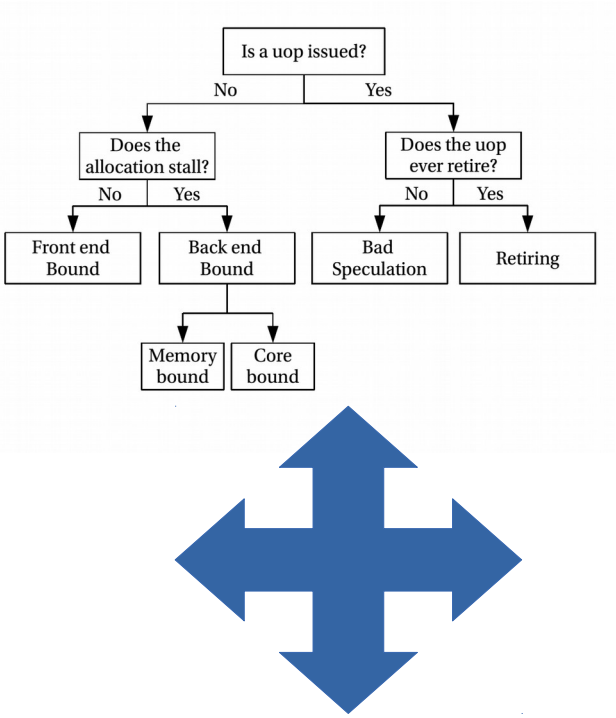
Dr. Christopher Dahnken
Intel GmbH

Outline

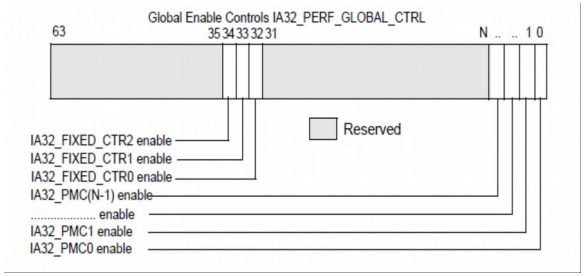
CPU



Method



Measurement



Code

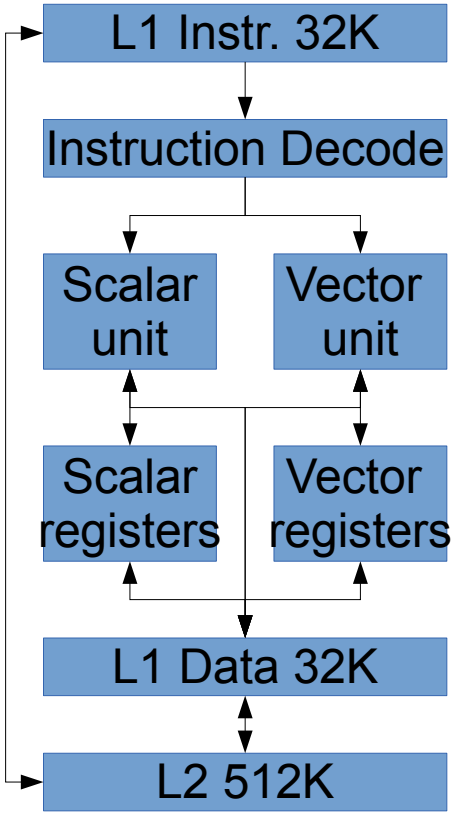
```
!$OMP SECTION
 if(iblock.lt.(nblocks)) then
   nxti=m_of_i(iblock+1)
   nxtj=n_of_i(iblock+1)
   nxtk=k_of_i(iblock+1)

   nxt_buffsize_m=buffersize(ms,bm,nxti)
   nxt_index_m=(nxti-1)*bm+1

   nxt_buffsize_n=buffersize(ns,bn,nxtj)
   nxt_index_n=(nxtj-1)*bn+1

   nxt_buffsize_k=buffersize(ks,bk,nxtk)
   nxt_index_k=(nxtk-1)*bk+1
```

The Xeon Phi Coprocessor

Intro

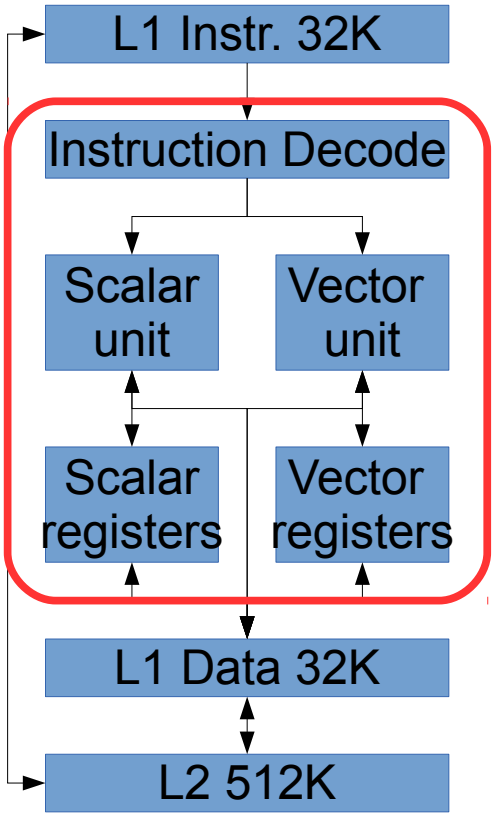
- In the very first part of this lecture, we have introduced many features that make modern CPU cores fast and easy program
- Now we will speak about an architecture, which removes some of these features in order to optimize size and power consumption, but then scales the core count to a much higher number.
-

KNC Core



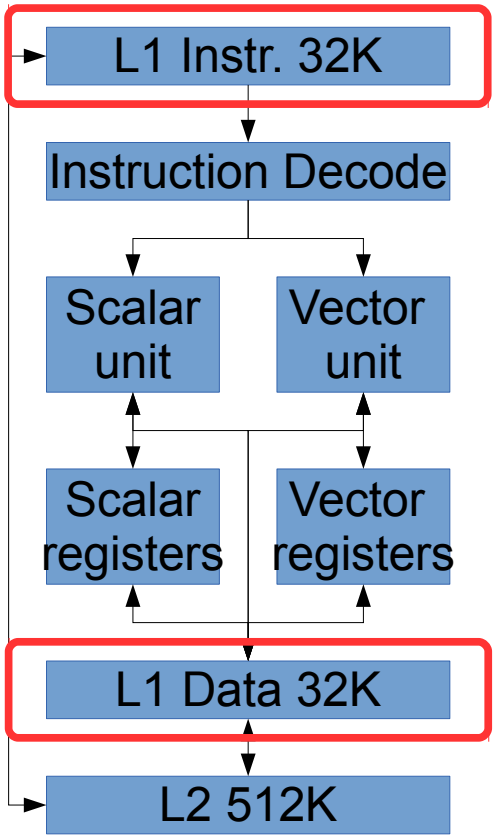
- Pentium scalar instruction set (x87!), fully functional
- In order-operation
- Full 64bit addressing
- 512bit vector unit
- 4 HW threads/core
- Two pipelines:
  - Scalar
  - Vector/Scalar

KNC Core



- 2 issue (1 scalar/1 vector)
- 2 cycle decoder: no-back to back cycle issue from the same context (thread)
- Most vec instructions have 4 clock latency
- At least two HW contexts (thread/proc) to fully utilize the core

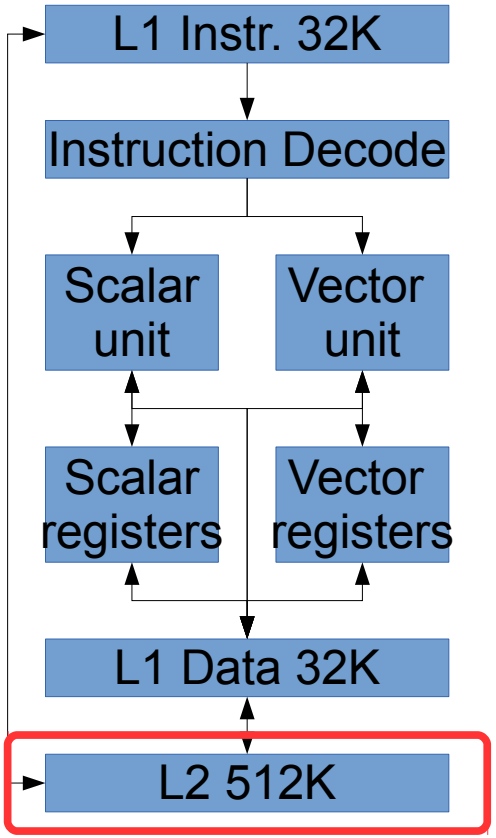
KNC Core



L1 caches

- 32K I-cache per core
- 32K D-cache per core
- 8 way associative
- 64byte cache line
- 3 cycle access latency
- Up to 8 outstanding requests
- Fully coherent (MESI)

KNC Core

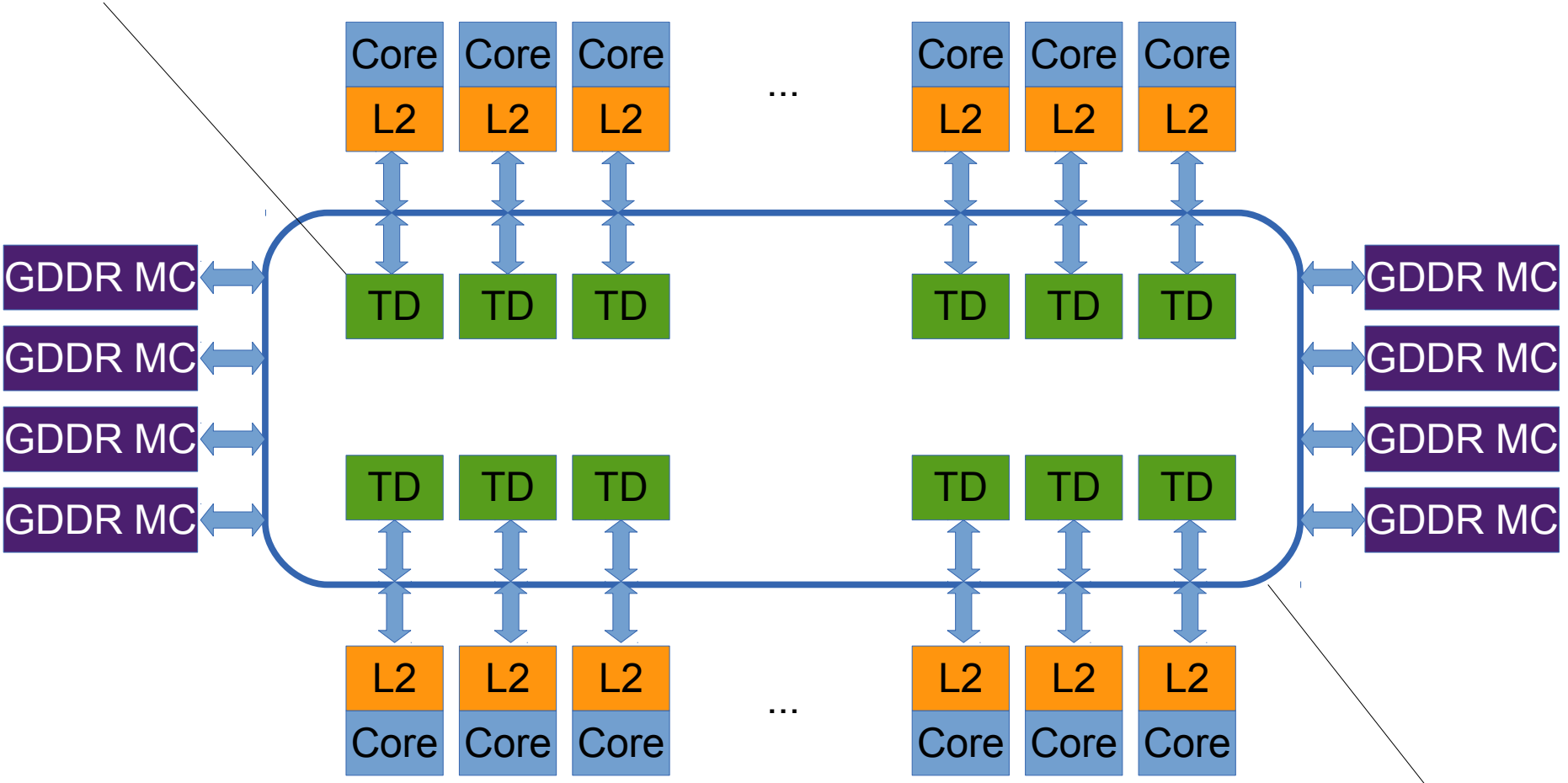


L2 cache

- 512K Unified per core
- 8 way assoc
- Inclusive
- 31M total across 62 cores
- 11 cycle raw access latency
- Up to 32 outstanding requests
- Streaming HW prefetcher
- Fully coherent

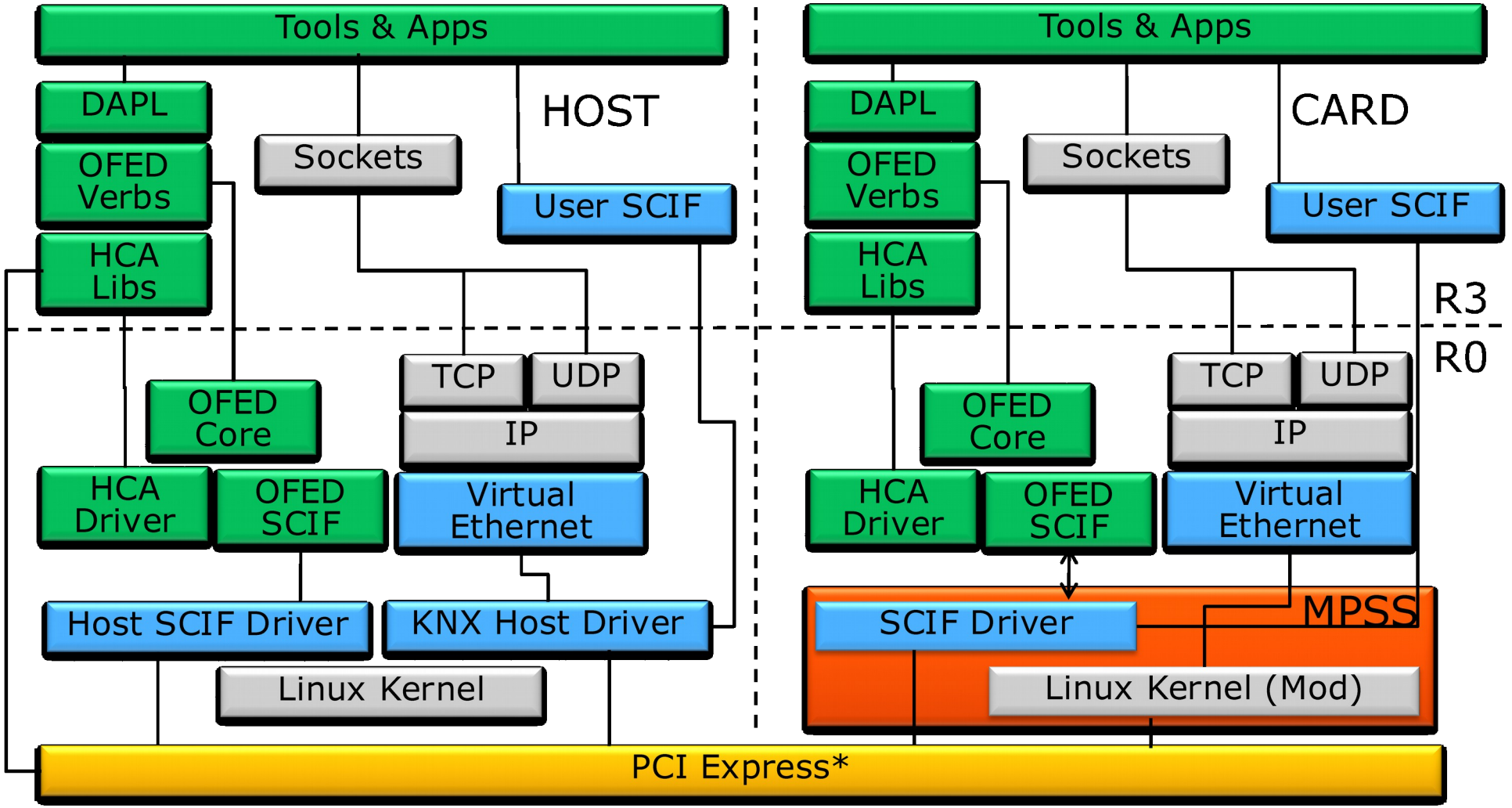
KNC Ring

Tag Directory
Track cache lines in all L2s.



Bi-directional ring
- 64byte data
- Command & address
- Coherence & Credits

KNC software architecture



Programming Xeon Phi

Native

- Compile for native architecture (-mmic)
- Run directly on the platform, or via MPI over many KNC or mixed Xeon/Xeon Phi environments

Offload

- Execution of the main program on the host
- Selected regions are executed on the coprocessor, along with required memory transfer

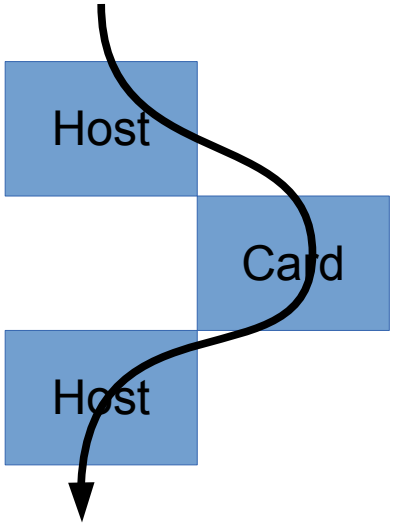
Here, we will only consider offload

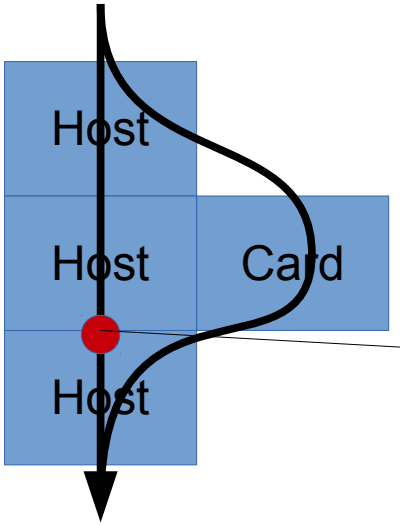
Programming Xeon Phi

Pragmas for offloading

Pragma for synchronous [asynchronous] offload	Syntax
Offload region	#pragma offload [signal(tag)]
Memory transfer and allocation (no code execution)	#pragma offload_transfer [signal(tag)]
Wait for signal (only for asynchronous mode)	#pragma offload_wait wait(tag)

Synchronous:
Host blocks while card is executing





Asynchronous:
Host execution continues while card is executing. Need a synchronisation point

Programming Xeon Phi

#pragma offload <clauses>

Clause	Syntax	Comment
Target specification	target(name[:card_number])	Where to run construct
Conditional offload	if (condition)	Boolean expression
Inputs	in(var-list modifiersopt)	Copy from host to coprocessor
Outputs	out(var-list modifiersopt	Copy from coprocessor to host
Inputs & outputs	inout(var-list modifiersopt	Copy host to coprocessor and back when offload completes
Non-copied data	nocopy(var-list modifiersopt)	Data is local to target

Programming Xeon Phi

Modifiers to in, out and inout

Modifiers	Syntax	Comment
Specify pointer length	length(element-count-expr)	Copy N elements of the pointer's type
Control pointer memory allocation	alloc_if (condition)	Allocate memory to hold data referenced by pointer if condition is TRUE
Control freeing of pointer memory	free_if (condition)	Free memory used by pointer if condition is TRUE
Control target data alignment	align (expression)	Specify minimum memory alignment on target

Programming Xeon Phi

Asynchronous offload – the signal clause

Clause	Syntax	Comment
Place a signal	signal(int)	Copy N elements of the pointer's type
Control pointer memory allocation	alloc_if (condition)	Allocate memory to hold data referenced by pointer if condition is TRUE
Control freeing of pointer memory	free_if (condition)	Free memory used by pointer if condition is TRUE
Control target data alignment	align (expression)	Specify minimum memory alignment on target

Programming Xeon Phi

Parallel reduction over all cores

```
double* arr=new double[len];
double ret=0;
#pragma offload target(mic:0) in(arr:length(len)) out(ret)
{
#pragma omp parallel for reduction(+:ret)
   for(int i=0;i<len;i++)
   {
      ret+=arr[i]
   }
}
```

The array is allocated, transferred and freed after the offload region completed. Only ret is transferred back.

Programming Xeon Phi

Sample pragma patterns

- In the following we will go though a number of example constructs to deal with memory management and execution
- We will assume that for each array on Xeon Phi, there is a host array present. There are way to do it without, but that is out of scope here.

Programming Xeon Phi

Allocation and deallocation of Xeon Phi memory

Explicitly allocating an array on Xeon Phi

```
#pragma offload_transfer \\
    nocopy(arr:length(len) alloc_if(1) free_if(0))
```

Explicitly deallocating an array on Xeon Phi

```
#pragma offload_transfer \\
    nocopy(arr:length(len) alloc_if(0) free_if(1))
```

Programming Xeon Phi

Transfer of data to and from Xeon Phi

Transfering data into an existing array on Xeon Phi

```
#pragma offload_transfer \\
    in(arr:length(len) alloc_if(0) free_if(0))
```

Transfering data from an existing array on Xeon Phi

```
#pragma offload_transfer \\
    out(arr:length(len) alloc_if(0) free_if(0))
```

Programming Xeon Phi

Asynchronous computation and sync wait

Assigning a value to the signal

The signal is an int or long immediate and should be unique for a particular array. In principle there is no constraints on the value.
A good choice is to initialize its value with the array address:

```
double* arr = new double[size];
long signal1=arr;
```

Asynchronous offload region

```
#pragma offload inout(arr:length(len)) signal(s)

 //… do something useful on the host

#pragma offload_wait signal(s)
```

Programming Xeon Phi

Asynchronous data transfer and sync wait

Transfering data into an existing array on Xeon Phi

```
#pragma offload_transfer \\
    in(arr:length(len) alloc_if(0) free_if(0))\\
    signal(s)
 //… do something useful on the host
#pragma offload_wait signal(s)
```

Transfering data from an existing array on Xeon Phi

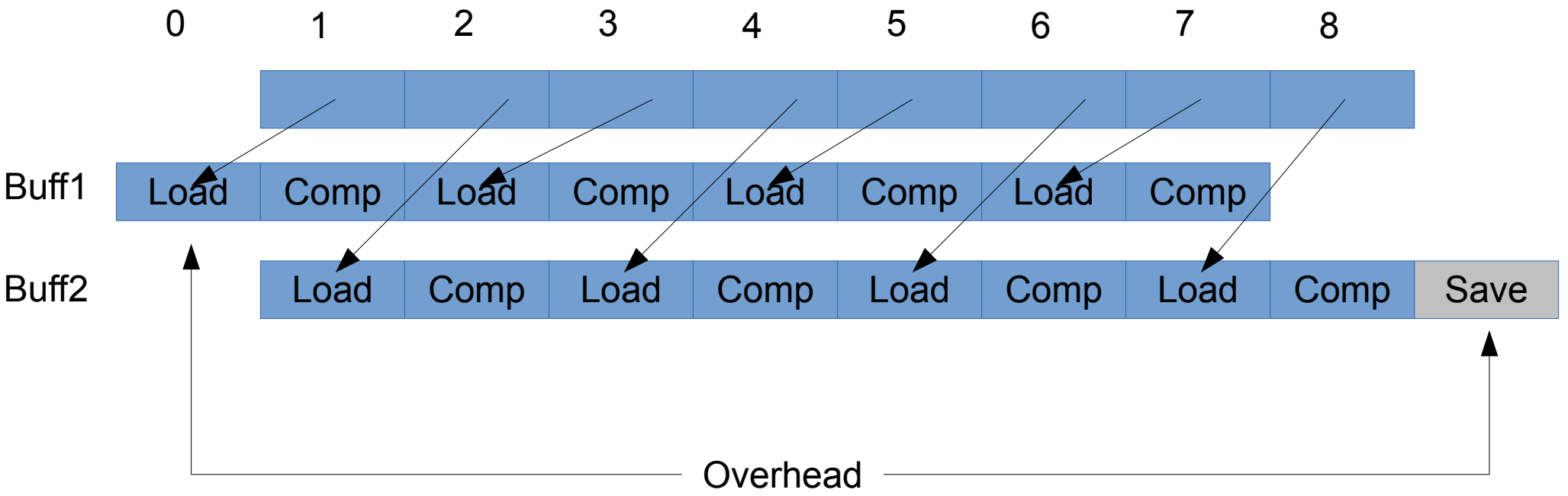
```
#pragma offload_transfer \\
    out(arr:length(len) alloc_if(0) free_if(0))\\
    signal(s)
 //… do something useful on the host
#pragma offload_wait signal(s)
```

Managing Memory

Programming Xeon Phi

- With Xeon Phi offloading, we have to keep in mind that we need to cross the PCI when transferring data to the card
- In order not to lose time in the transfer process, it is advisable to keep data allocated if possible
- Also, when an algorithm can be blocked, data transfer should be overlapped with computation

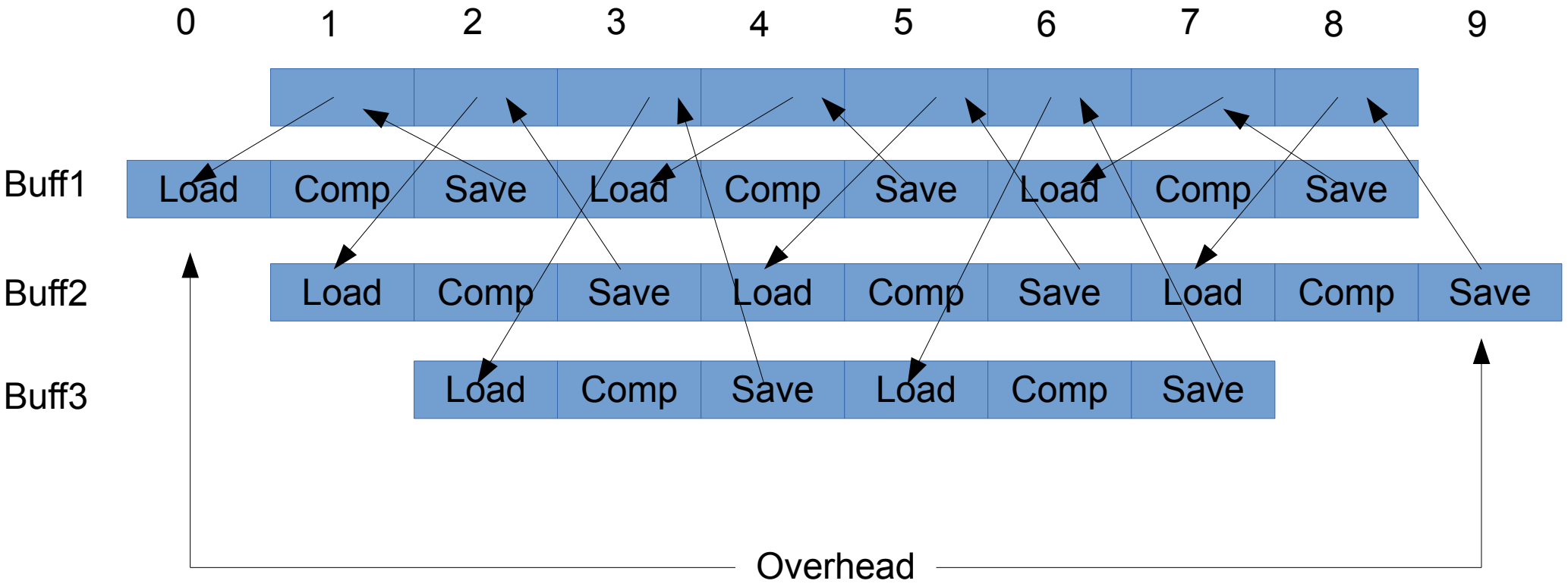
Mem transfers - Double buffers



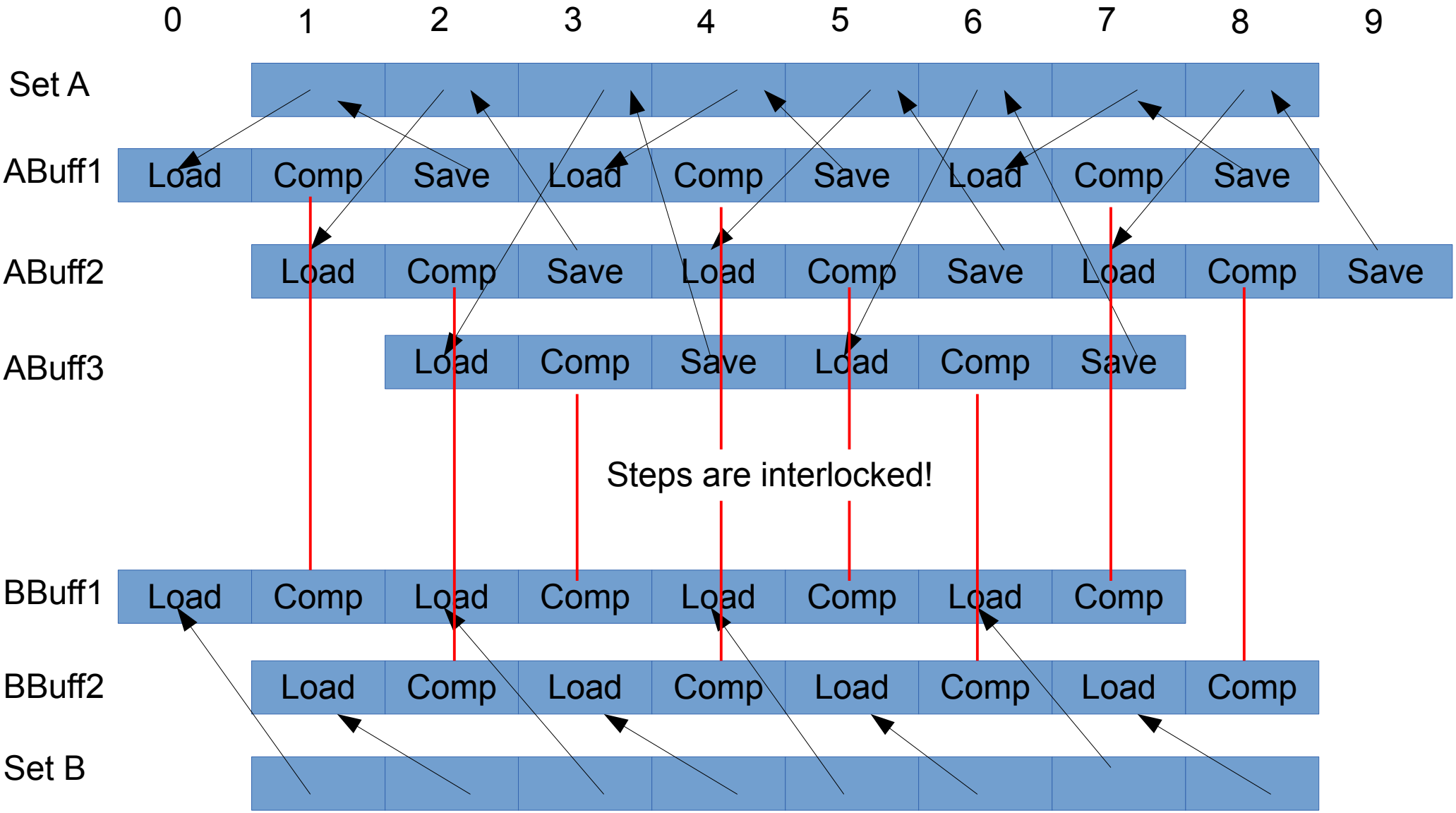
Double buffers are a consumer scheme – there is no safe write back unless it is synchronous (e.g. at the end)

Mem transfers - tripple buffers

When working with buffers, there is usually more than enough space on the card to do some fancy stuff. Tripple buffers allow for safe write back.



Double and Tripple Buffers

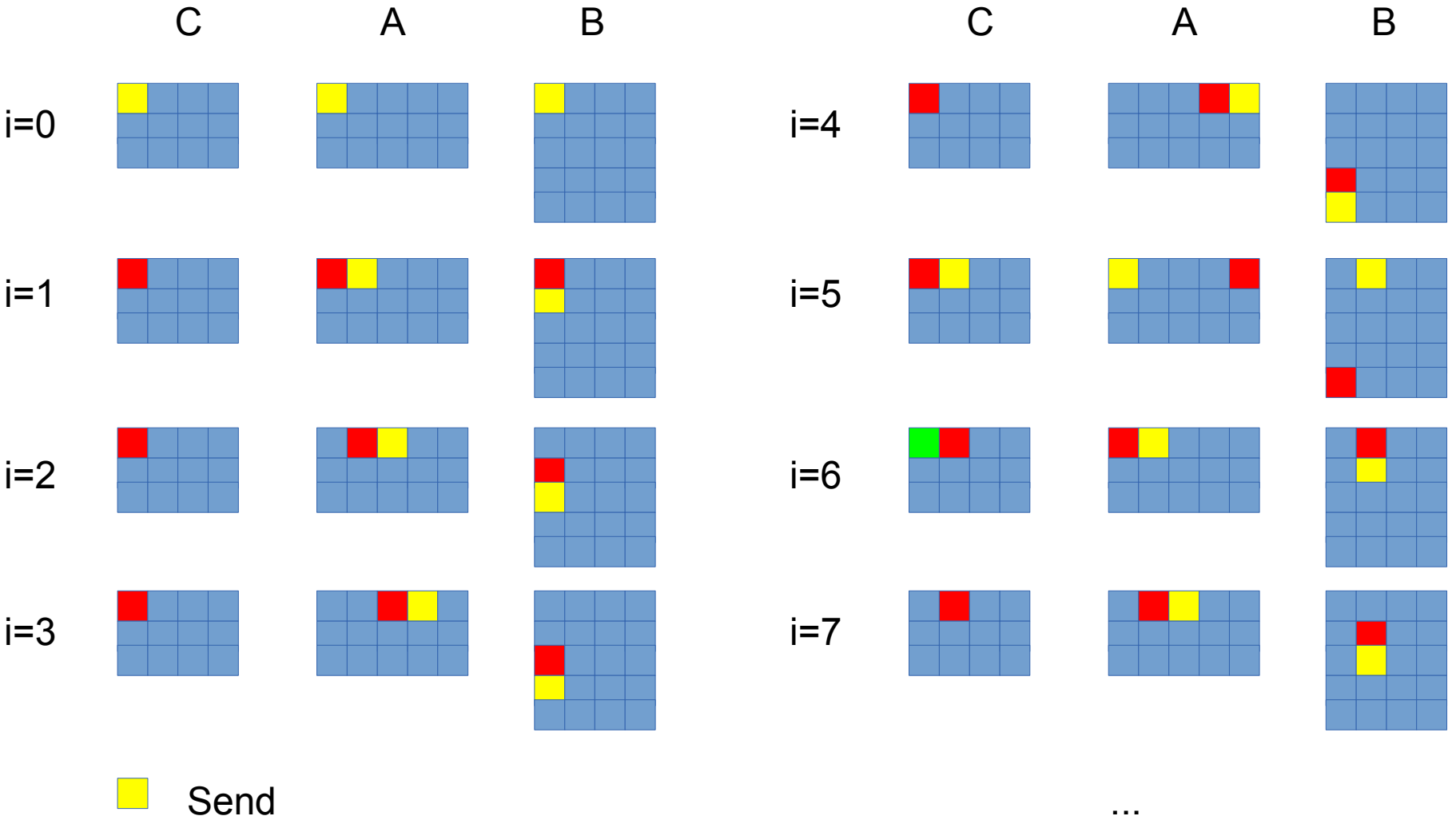


Case Study – Quantum Espresso

- Main hotspots are: Complex linear algebra (ZGEMM), FFT and eigensolvers
- Here we will look at ZGEMM
- High performance methods are available in MKL
- The remaining issue is only the memory management

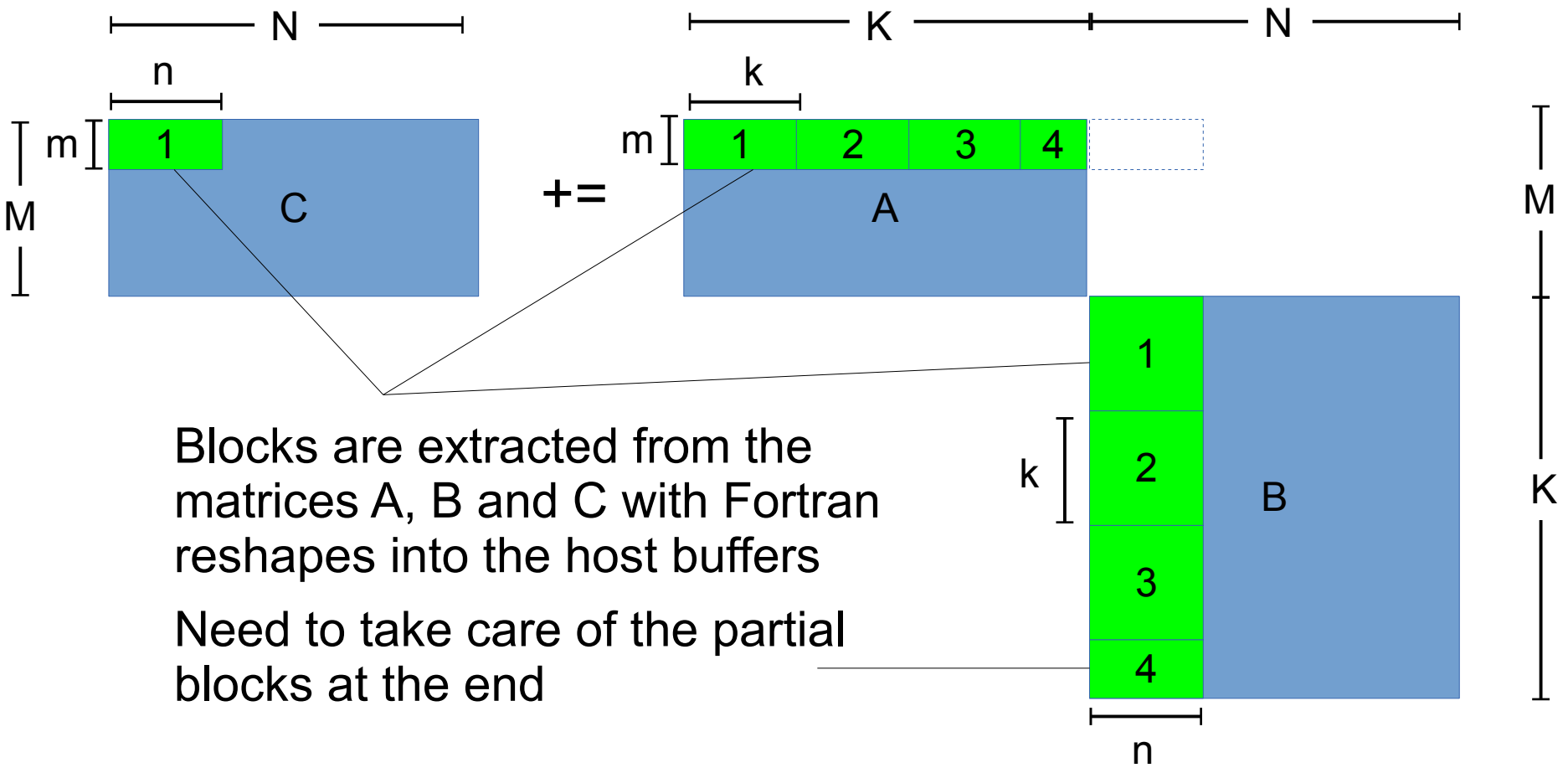
Offloading ZGEMM

Offloaded matrix muliplication (XGEMM)



Offloading ZGEMM

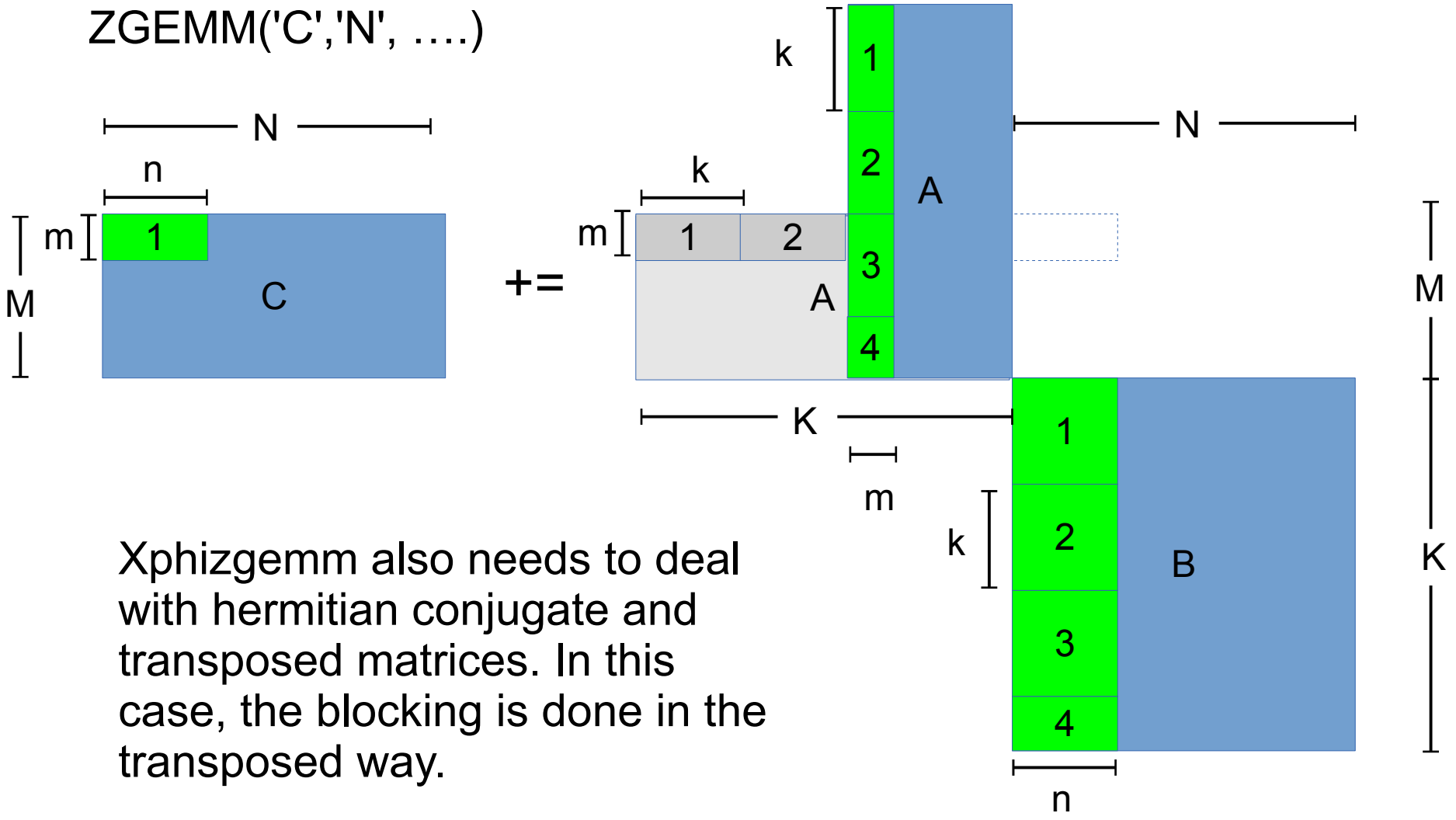
ZGEMM('N','N', ….)



Blocks are extracted from the matrices A, B and C with Fortran reshapes into the host buffers

Need to take care of the partial blocks at the end

Offloading ZGEMM



Xphizgemm also needs to deal with hermitian conjugate and transposed matrices. In this case, the blocking is done in the transposed way.

Pipeline Algorithm

```
for(int i=-1;i<length+1;i++){
if(i<length){getBlocks(i+1); sendBlocksToCard(i+1);}
if(i>1){sendBlocksToHost(i-1); setBlock(i-1);}
if(i>-1 && i<length) ComputeOffload(i);
}
```

.

Vectorization

Xeon Phi Vectorization

- We have already talked about vectorization (implicit and explicit) when treating AVX
- Xeon Phi has SIMD vectors that are 512bit wide (twice as wide as AVX)
- The instruction set is quite similar in functionality
- Here we will show a case study of it's usage in CP2K

Xeon Phi Vectorization

- CP2K uses matrices of the size of the number of base functions used in their representation in density function theory
- The most common size is 23x23 (Water)
- Many of these multiplication can be performed in parallel

Small Matrix-Matrix Multiplication 2

- Analysis of MKL DGEMM performance shows that for small matrices it underperforms against an individual implementation
- A wrapper library was created, having an individual unrolling for symmetric matrices <32x32
- For larger (or non-symmetric cases) fall back to MKL

Small Matrix-Matrix Multiplication 3

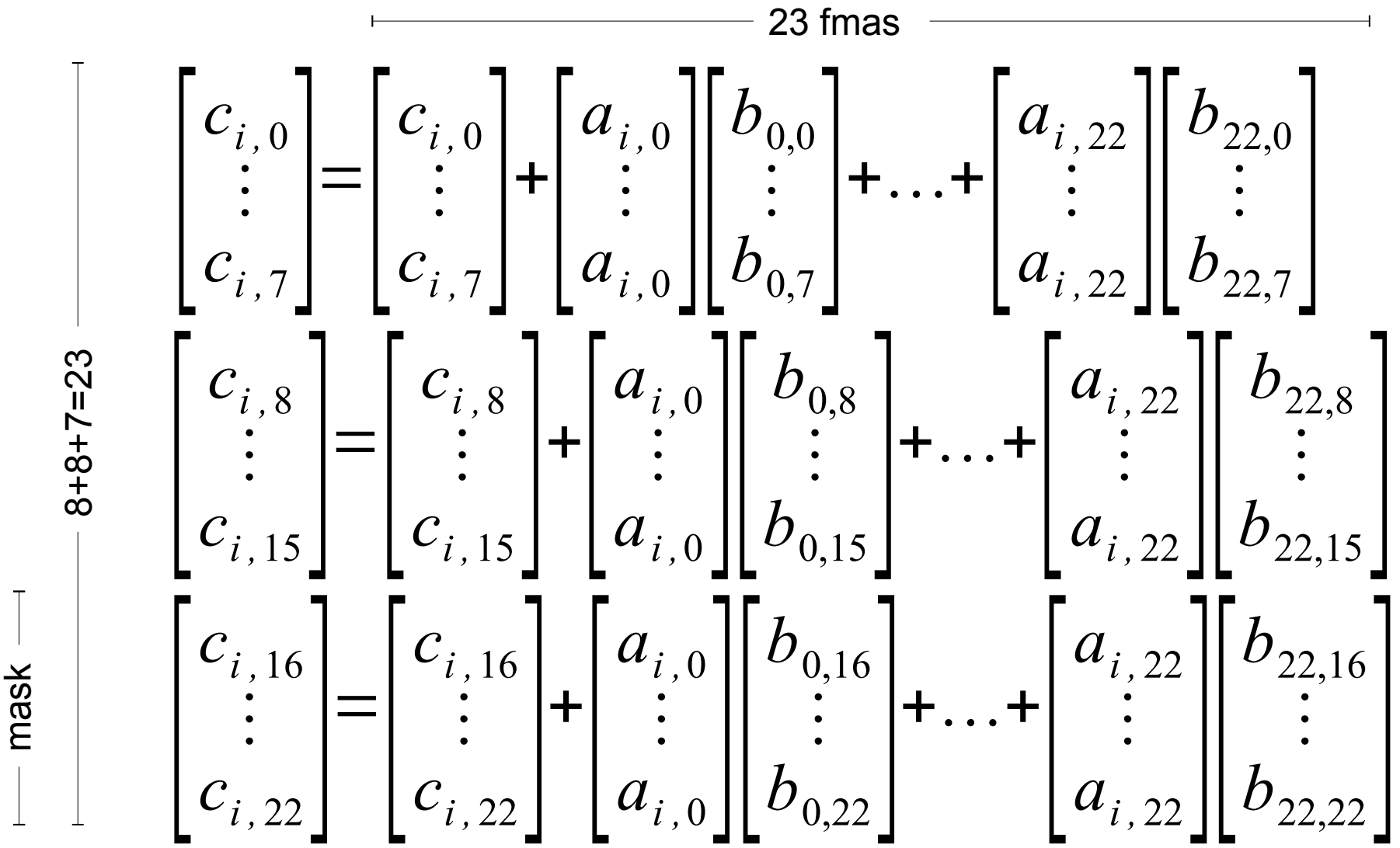
compiler vectorization

- Example of simple workaround (first try)

```
SUBROUTINE MICMM_23(C,A,B)
IMPLICIT NONE
DOUBLE PRECISION, INTENT(IN) :: A(23,23)
DOUBLE PRECISION, INTENT(IN) :: B(23,23)
DOUBLE PRECISION, INTENT(INOUT) :: C(23,23)
INTEGER :: I,J
!DEC$ SIMD
DO i=1,23
  DO j=1,23
    C(i,j)=C(i,j)+A(i,1)*B(1,j)+A(i,2)*B(2,j)+A(i,3)*B(3,j)+A(i,4)*B(4,j)+A(i,5)*B(5,j)
      +A(i,6)*B(6,j)+A(i,7)*B(7,j)+A(i,8)*B(8,j)+A(i,9)*B(9,j)+A(i,10)*B(10,j)
      +A(i,11)*B(11,j)+A(i,12)*B(12,j)+A(i,13)*B(13,j)+A(i,14)*B(14,j)
      +A(i,15)*B(15,j)+A(i,16)*B(16,j)+A(i,17)*B(17,j)+A(i,18)*B(18,j)
      +A(i,19)*B(19,j)+A(i,20)*B(20,j)+A(i,21)*B(21,j)+A(i,22)*B(22,j)+A(i,23)*B(23,j)
  ENDDO
ENDDO
END SUBROUTINE
```

Small Matrix-Matrix Multiplication 4

intrinsics implementation



Small Matrix-Matrix Multiplication 5

Example for 3x3 MM with intrinsics

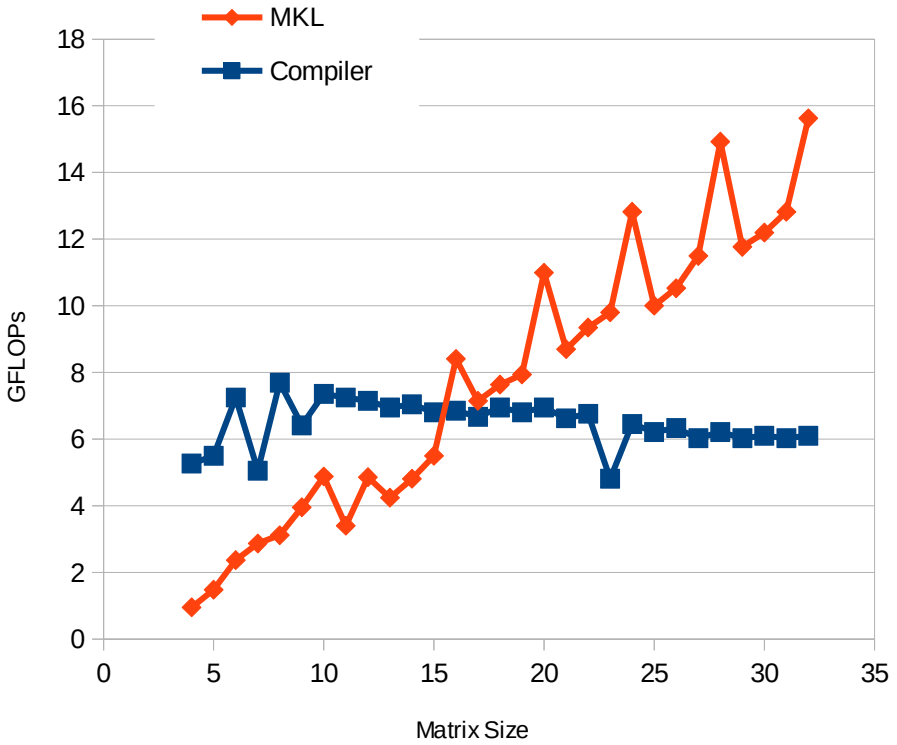
```
void micgemm_3_3_3(double* a,double* b,double* c){
int i;
__m512d xa0;
__m512d xa1;
__m512d xa2;
__m512d xb0;
__m512d xb1;
__m512d xb2;
__m512d xc0;
 xb0 = _MM512_MASK_LOADU_PD(&b[0+0],7);
 xb1 = _MM512_MASK_LOADU_PD(&b[3+0],7);
 xb2 = _MM512_MASK_LOADU_PD(&b[6+0],7);
for(i=0;i<9;i+=3){
    xc0 = _MM512_MASK_LOADU_PD(&c[i+0],7);
    xa0=_mm512_set1_pd(a[i+0]);
    xa1=_mm512_set1_pd(a[i+1]);
    xa2=_mm512_set1_pd(a[i+2]);
    xc0=_mm512_fmadd_pd(xa0,xb0,xc0);
    xc0=_mm512_fmadd_pd(xa1,xb1,xc0);
    xc0=_mm512_fmadd_pd(xa2,xb2,xc0);
    _MM512_MASK_STOREU_PD(&c[i+0],xc0,7);
}
```

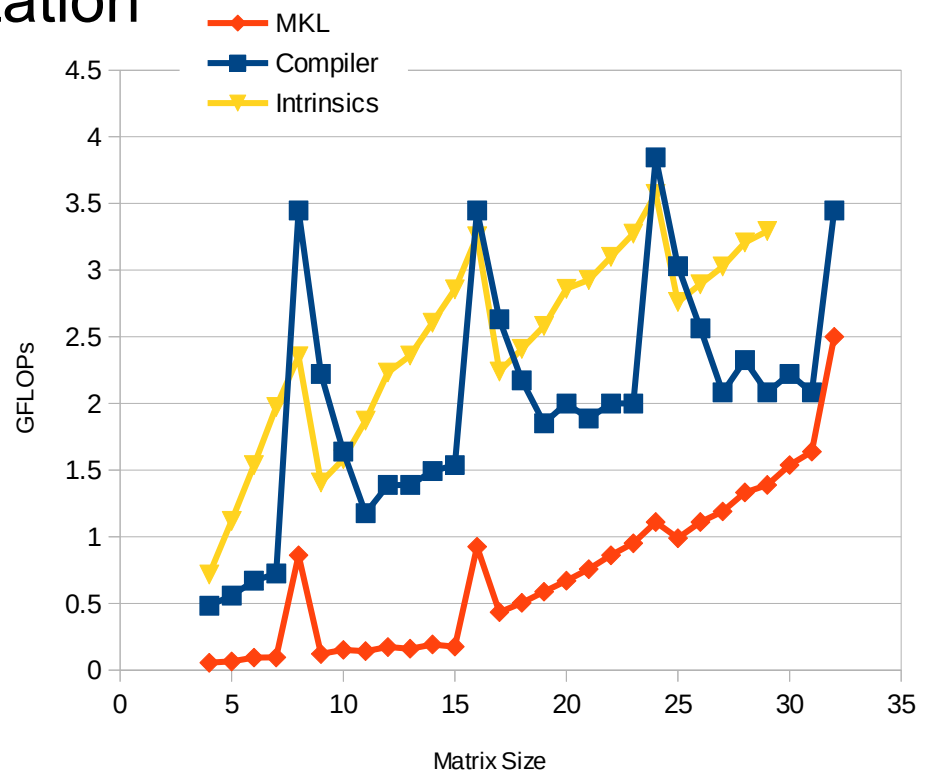
Small Matrix-Matrix Multiplication 6

SNB

Compiler/Intrinsics vectorization

KNC





Relevant KNC data:
MKL DGEMM 23x23: 0.95 GFLOPs
Compiler DGEMM 23x23: 1.99 GFLOPs
Intrinsics DGEMM 23x23: 3.33 GFLOPs
Evaluated over 1TFLOP total, single core

3.5x better for 23x23

Summary

- Xeon Phi is a coprocessor with >TFLOPs performance and up to 61 cores
- It can be programmed natively or in an offload fashion
- The offload interface is an easy pragma based model
- High levels of performance can be achieved, but high vectorization ad parallelization is required.