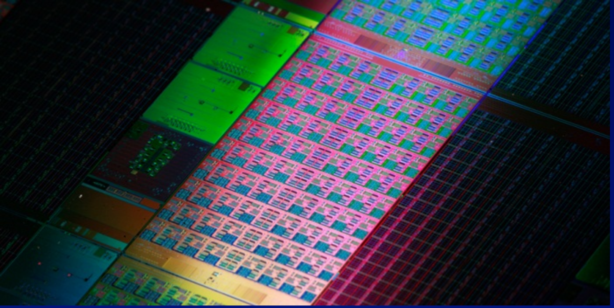


SIMD: CilkPlus and OpenMP

Kent Milfeld, Georg Zitzlsberger, Michael Klemm, Carlos Rosales

ISC15
The Road to Application Performance on Intel Xeon Phi
July 16, 2015

THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

SIMD

SIMD

What to consider about SIMD

- Know as Vectorization by scientific community.
- Speed Kills
  - It was the speed of microprocessors that killed the Cray vector story in the 90’s.
  - We a rediscovering how to use vectors.
  - Microprocessor vectors were 2DP long for many years.
- We live in a parallel universe
  - It’s not just about parallel SIMD, we also live in a silky environment of thread tasks and MPI tasks.

What to make of this?

- SIMD registers are getting wider now, but there are other factors to consider.
  - Caches: Maybe non-coherent, possible 9 layers of memory later
  - Alignment: Avoid cache-to-register hickups
  - Prefetching: MIC needs user intervention here
  - Data Arrangement: AoS vs SoA, gather, scatter, permutes
  - Masking: Allows conditional execution– but you get less bang for your buck.
  - Striding: 1 is best

SIMD

Intel CilkPlus

- # pragma SIMD
  - Force SIMD operation on loops
- Array Notation → data arranged appropriate for SIMD

a[index:count]	start at index, end count-start-1
	(also index:count:stride)
a[i:n] = b[i-1:n]+b[i+1]	(think single line SIMD, heap arrays)
	(optimize away subarrays)
e[:] = f[:] +g[:]	(entire array, heap or stack)
r[:] = s[i[:]], r[i[:]=s[:]	(gather, scatter)
func(a[:])	(scalar/simd-enabled=by element/SIMD)
if(5==a[:]) result[:]=0	(works with conditionals)

- SIMD Enabled Functions: element →vector function

SIMD pragma

- Instructs the compiler to create SIMD operations for iterations of the loops.
- Reason for vectorization failure: too many pointers, complicated indexing … (ivdep is a hint)

Without pragma vec-report=2 was helpful:

remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

```
void do2(double a[n][n], double b[n][n], int end){
#pragma SIMD
for (int i=0 ; i<end ; i++) {
    a[i][0] = (b[i][0] - b[i+1][0]);
    a[i][1] = (b[i][1] - b[i+1][1]);
  }
}
```

ivdep and vector always don't work here.

(Fortran code vectorizes)

SIMD Enabled Functions

- SIMDizable Functions:

```
double fun1(double r, double s, double t);
double fun2(double r, double s, double t);

…

void driver (double R[N], double S[N], double T[N]){
  for (int i=0; i<N; i++){
    A[i] = fun1(R[i],S[i],T[i]);
    B[i] = fun2(R[i],S[i],T[i]);
  }
}
```

courier

SIMD Enabled Functions

- Can be invoked with scalar or vector arguments.
- Use array notation with SIMD version (optimized for vector width)

**

```
__declspec(vector) double fun1(double r, double s, double t);
__declspec(vector) double fun2(double r, double s, double t);

        // Function is for an element operation;
...     // but in parallel context (CilkPlus) provides an array for a vector version.

void driver (double R[N], double S[N], double T[N]){
    A[:] = fun1(R[:],S[:],T[:]);
    B[:] = fun2(R[:],S[:],T[:]);
}
```

** or __attribute((vector))

SIMD Enabled Functions

- Vector attribute/declspec decorations generate scalar and SIMD version with:

Syntax:

__attribute__((vector (clauses))) *function_declaration*
__declspec(vector(clauses)) *function_declaration*

Clauses:

vectorlength(n)	Vector Length
linear(*list : step*)	scalar list variables are incremented by step;
uniform(*list*)	(same) values are broadcast to all iterations
[no]mask	generate a masked vector version

SIMD and Threads

- Cilk’s “los tres amigos”
  - cilk_for
  - cilk_spawn
  - cilk_sync
- Cilk loops are SIMDizes, and invoke multiple threads.
- Functions use SIMD form in CilkPlus loops.

SIMD

OpenMP SIMD

- First appeared in OpenMP 4.0   2013
- Appears as
  - SIMD
  - SIMD do/for
  - declare SIMD

- SIMD refinements in OpenMP 4.1, ~2015.

SIMD

- OMP Directive SIMDizes loop

Syntax (Fortran):

!$OMP SIMD [*clause*[[,] *clause*] ...]
#pragma omp SIMD [*clause*[[,] *clause*] ...]

Clauses:

safelen(*n*)	number (n) of interations in a SIMD chunk
linear(*list : step*)	scalar list variables are incremented by step; loop iterations incremented by (vector length)*step
aligned(*list* :*n*)	uses aligned (by n bytes) move on listed variables

collapse(*n*), lastprivate(*list*), private(*list*), reduction(*operator*: *list*)

SIMD + Worksharing Loop

- OMP Directive Workshares and SIMDizes loop

Syntax:

```
!$OMP DO      SIMD [clause[[,] clause] ... ]
#pragma omp SIMD [clause[[,] clause] ... ]
```

Clauses:

any DO clause   data sharing attributes, nowait, etc.
any SIMD clause

Creates SIMD loop which uses chunks containing increments of the vector size.
Remaining iterations are distributed “consistently”.
No scheduling details are give.

SIMD Enabled Functions

- OMP Directive generates scalar and SIMD version with:

Syntax (Fortran):

$OMP DECLARE SIMD(*routine-name*) [*clause*[[,] *clause*]...]

Clauses:

aligned(*list*:*n*)	uses aligned (by n bytes) moves on listed variables
[not]inbranch	must always be called in conditional [or never in]
linear(*list:step*)	scalar list variables are incremented by step;
	loop iterations incremented by (vector length)*step
simdlen(*n*)	vector length
uniform(*list*)	listed variables have invariant value

SIMD

CilkPlus → OpenMP Mapping

CilkPlus

- SIMD (on loop)
  - Reduction
  - Vector length
  - Linear (increment)
  - Private, Lastprivate

OpenMP

- SIMD (on loop)
  - Reduction
  - Vector length
  - Linear (increment)
  - Private, Lastprivate

e.g (fortran)

```
!dir$ simd reduction(+:mysum) linear(j:1) vectorlength(4)
      do…; mysum=mysum+j; j=fun(); enddo
```

```
!$omp simd reduction(+:mysum) linear(j:1) safelen(4)
      do…; mysum=mysum+j; j=fun(); enddo
```

CilkPlus OpenMP SIMD Differences

CilkPlus

- SIMD (on loop)
  - firstprivate
  - vectorlengthfor
  - [no]vectremainder
  - [no]assert
- #pragma cilk grainsize

OpenMP

- SIMD (on loop)
  - aligned(var_list,bsize)
  - collapse
  - schedule(kind, chunk)

CilkPlus Enable OMP Declare Differences

CilkPlus

- vector clauses
  - vectorlength
  - linear
  - uniform
  - [no]mask

  - processor(cpuid)
  - vectorlengthfor

OpenMP

- declare simd
  - simdlen
  - linear
  - uniform
  - inbranch/notinbranch

  - aligned

TACC

Alignment

- Memory Alignment
  - Allocation alignment
    - C/C++
      - dynamic: memalloc routines
      - static: __declspec(align(64)) *declaration*
    - Fortran
      - dynamic: !dir$ attributes align: ***64*** :: *var*
      - static: !dir$ attributes align: ***64*** :: *var*
      - compiler: -align array64byte

Alignment (CilkPlus)

- Memory Alignment
  - Access Description
    - C/C++
      - loop: #pragma vector aligned (all variables)
      - Cilk_for vars: _assume_aligned(var,size)
      - pointers attribute: __attribute__((align_value (size)))
    - Fortran
      - dynamic: !dir$ attributes align: ***64*** :: *var* (allocatable var)
      - static: !dir$ attributes align: ***64*** :: *var* (stack var)
      - compiler: -align array64byte

Alignment (OpenMP)

- Memory Alignment
  - No API functions, no separate construct
  - Declaration SIMD / SIMD have aligned clauses

Prefetch

- Prefetch distance can be controlled via compiler options and pragmas

```
#pragma prefetch var:hint:distance
```

- inner loops
- may be important to turn off prefetch
- available for Fortran

What do developers need to control at the directive level?

- Caches: locality of data
- Alignment: Avoid cache-to-register hickups
- Prefetching: hiding latency (not available with OMP)
- Rearranging data: characterizing data structure (kokkos)
- Masking: Allows conditional execution– but you get less bang for your buck.
- Striding: characterized data structure, (restrict)

SIMD

Vector Compiler Options

- Compiler will look for vectorization opportunities at optimization
  - – O2 level.
- Use architecture option:
  –x<*simd_instr_set*> to ensure latest vectorization hardware/instructions set is used.
- Confirm with vector report:
  - – vec-report=<n>, n="verboseness"
- To get assembly code, *myprog*.s:
  - – S
- Rough Vectorization estimate: run w./w.o. vectorization
  -no-vec

Vector Compiler Options (cont.)

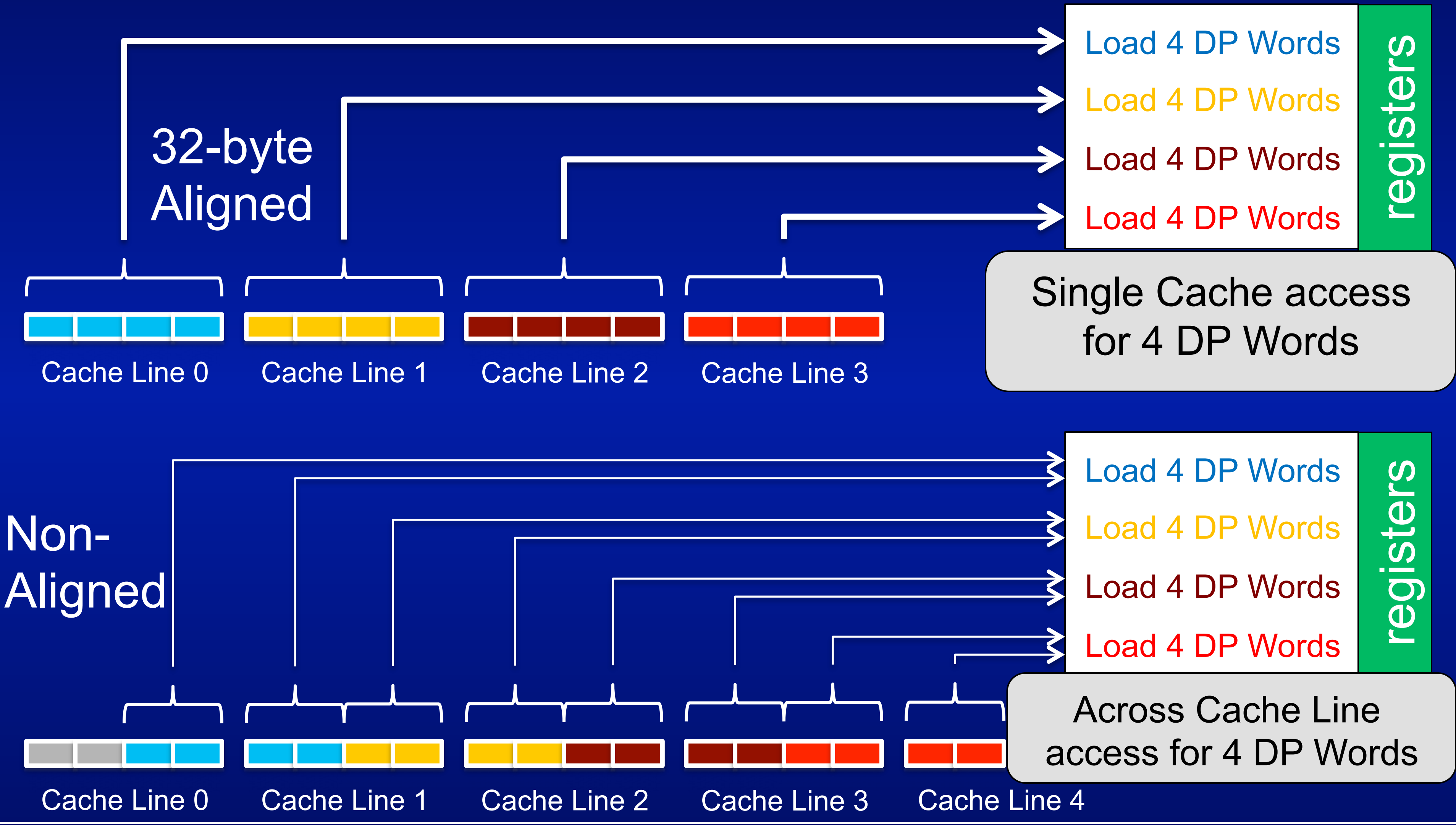
- Alignment options here.
- Inlining here.

Alignment

- Alignment of data and data structures can affect performance. For AVX, alignment to 32byte boundaries (4 Double Precision words) allows a single reference to a cache line for moving 4DP words into the registers (SIMD support). For MIC, alignment is 64 bytes.
- Compilers are great at detecting alignment and peeling off a few iterations before working on a sustained alignment within a loop body.

(Aligned data can use the more efficient movdqa instruction, rather than the less efficient movdqu instruction.)

Alignment


32-byte Aligned
Load 4 DP Words
Load 4 DP Words
Load 4 DP Words
Load 4 DP Words
registers
Cache Line 0
Cache Line 1
Cache Line 2
Cache Line 3
Single Cache access for 4 DP Words
Non-Aligned
Load 4 DP Words
Load 4 DP Words
Load 4 DP Words
Load 4 DP Words
registers
Cache Line 0
Cache Line 1
Cache Line 2
Cache Line 3
Cache Line 4
Across Cache Line access for 4 DP Words

Vector Align

- Unaligned accesses are slower.
  - Non-sequential across “bus”.
  - Cross cache line boundary.
- #pragma vector aligned or !DEC$ vector aligned

Alignment can be forced

C: memalign(XXbyte,size)

F90: Use compiler option
  -align arrayXXbyte

```
for(i=0; i<loops; i++)
  #pragma vector aligned
  for(j=0;j<N-i;j++) a[j]=b[j]+c[j];
```

```
0.75 CP/Op   w.o. pragma*
0.50 CP/OP    with pragma*
```

*When executed without –xSSE4.1 on Westmere.

Inlining

- Functions within a loop prevent vectorization.
  - Inlining can often overcome this problem.

  e.g.

file main.c

```
 ...
for(i=0; i<nx; i++){
     x = x0 + i*h;
     sum = sum + do_r2(x,y, xp,yp);
  }
...
```

file funs.c

```
double do_r2(double x, double y, double xp, double yp){
      double r2;
      r2 = (x-xp)*(x-xp) + (y-yp)*(y-yp);
      return r2;
}
```

- Since the call and function are in different files, inlining and vectorization don't occur. Use interprocedural optimization option (-ipo) to inline & vectorize.
- If call and function are within the same unit (file), inlining and vectorization are performed at –O2 optimization and higher.

Inlining

file main.c

```
…
for(i=0; i<nx; i++){
     x = x0 + i*h;
     sum = sum + do_r2(x,y, xp,yp);
  }
…
```

file funs.c

```
double do_r2(double x, double y, double xp, double yp){
  double r2;
  r2 = (x-xp)*(x-xp) + (y-yp)*(y-yp);
  return r2;
}
```

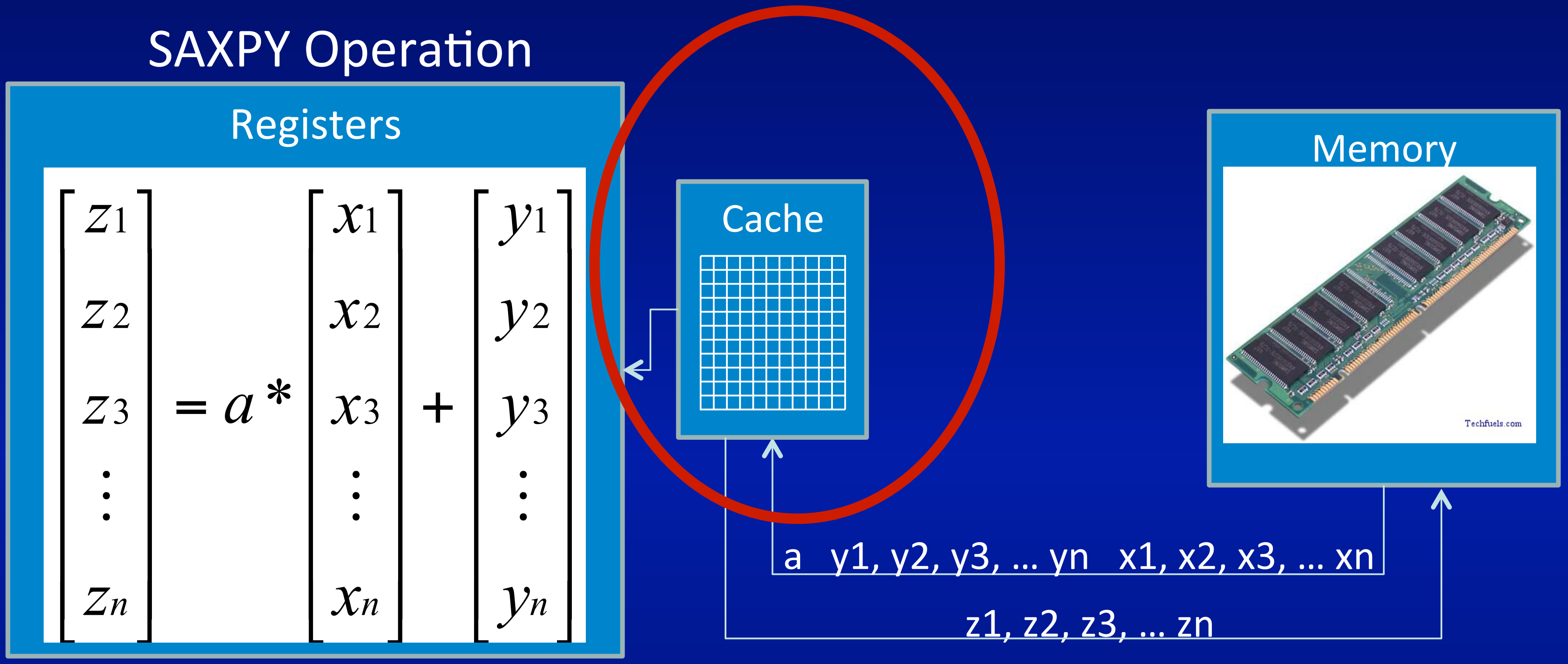
Inlining	Vectorization	Time (ms)
not inlined	not vectorized	1.55
inlined	not vectorized	0.44
inlined	vectorized	0.056

SIMD END

- Questions
- Discussion
- …

some Slides from 2012 Tutorial (kfm)

SIMD Hardware (for Vectorization)

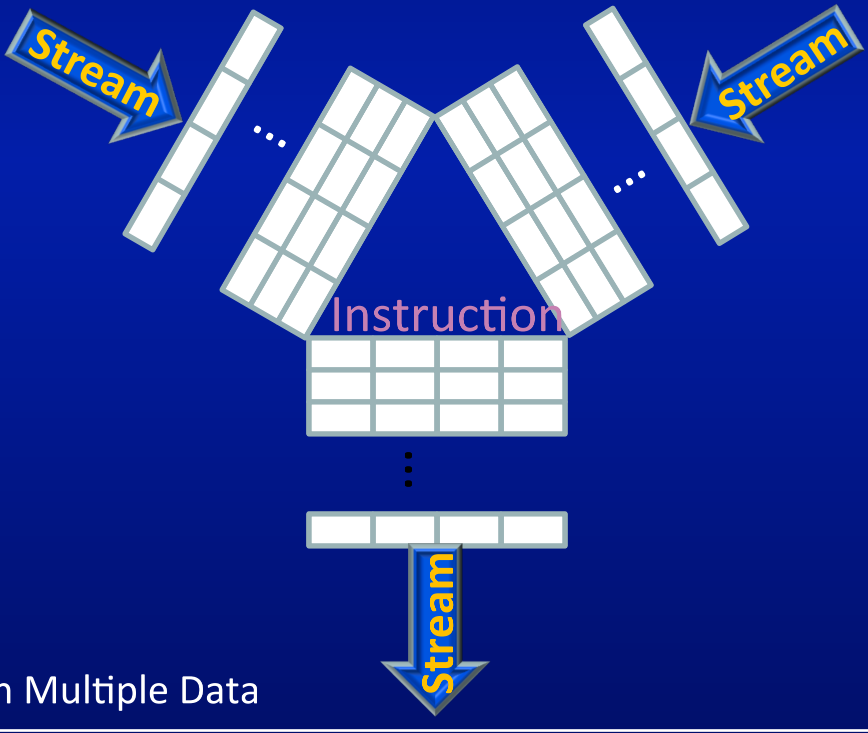


$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_n \end{bmatrix} = a * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}$$

- Optimal Vectorization requires concerns beyond the SIMD Unit!
  - Operations: Requires elemental (independent) operations (SIMD operations)
  - Registers: Alignment of data on 64, 128, or 256 bit boundaries might be important
  - Cache: Access to elements in caches is fast, access from memory is much slower
  - Memory: Store vector elements sequentially for fastest aggregate retrieval

SIMD Processing -- Vectorization

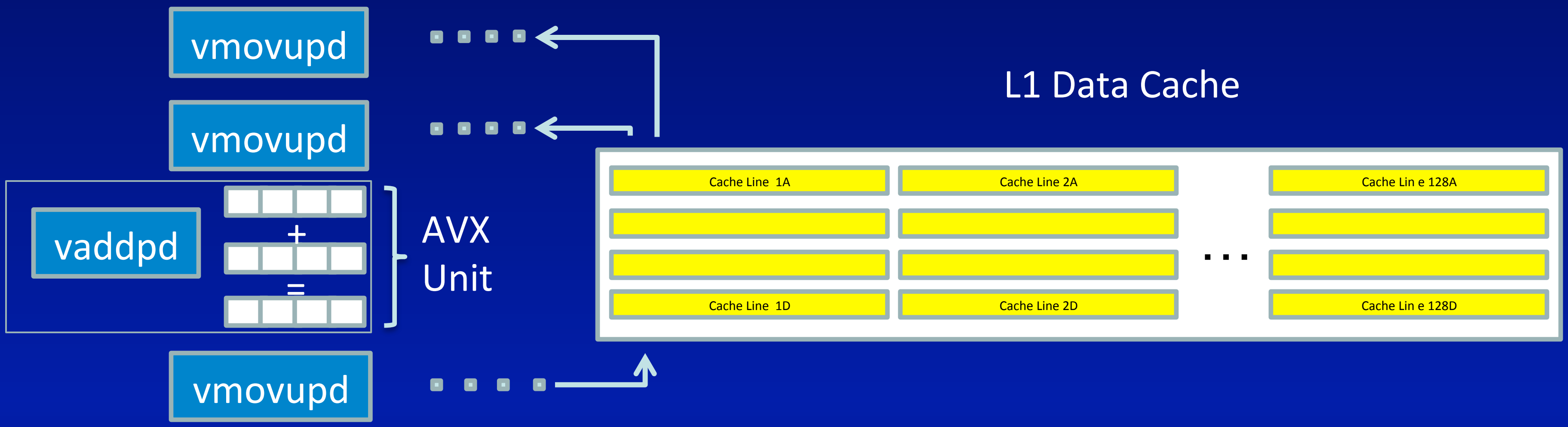
- Vectorization, or SIMD* processing, allows a simultaneous, independent instruction on multiple data operands with a single instruction. (Loops over array elements often provide a constant stream of data.)



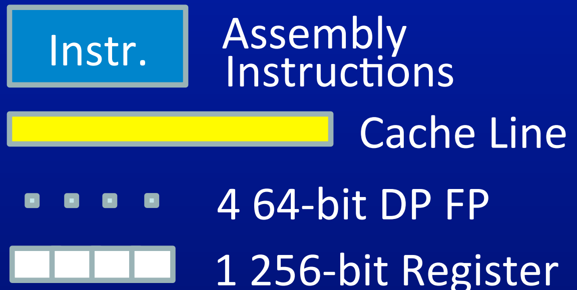
Note: Streams provide Vectors of length 2-16 for execution in the SIMD unit.

*SIMD= Single Instruction Multiple Data

Vector Add -- AVX



- Only vector code will load multiple sets of data into registers simultaneously.
- Non-aligned sets do consume more Clock Periods (CPs).



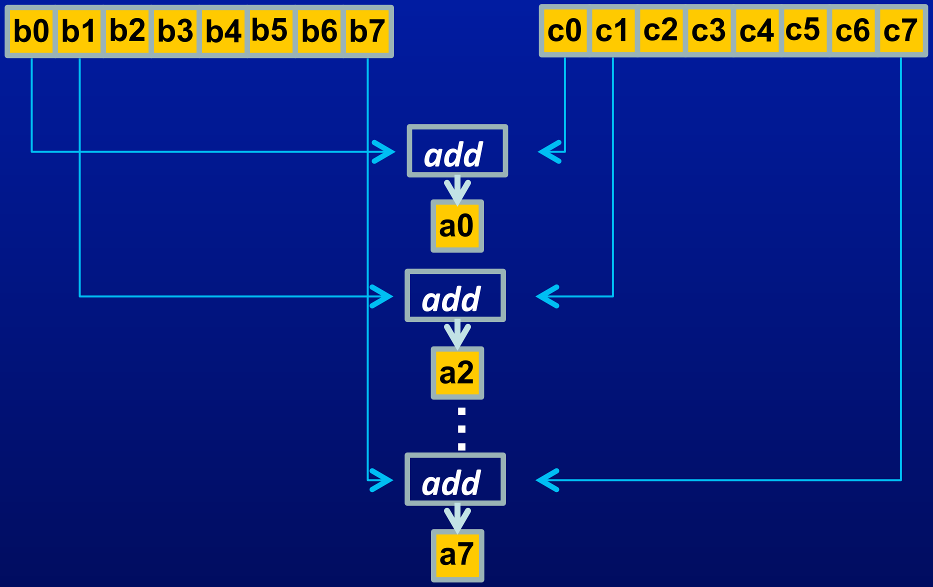
Vectorization (on KNC)

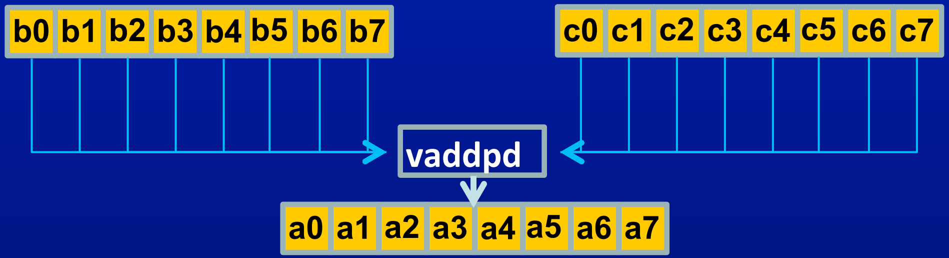
```
void mult(double *a, double *b, double*c, int n){
      for(int i=0; i<n; i++)  a[i]=b[i]+c[i];  }
```

```
subroutine mult(a, b, c, n); real*8 :: a(n),b(n),c(n)
      do i=1,n; a(i)=b(i)+c(i); enddo
end subroutine
```

Scalar Instructions
8 instructions, 8 element pairs

Vector Instruction
1 instruction, 8 element pairs





Compiler Directives: Hints and Coercion

alloc_section
distribute_point
inline, noinline, and forceinline
ivdep
loop_count
memref_control
novector
optimize
optimization_level
prefetch/noprefetch
simd
unroll/nounroll
unroll_and_jam/nounroll_and_jam
vector