Vectorization Quality: How Well is Your C Code Compiled?

Hiroshi Nakashima
(Kyoto University)

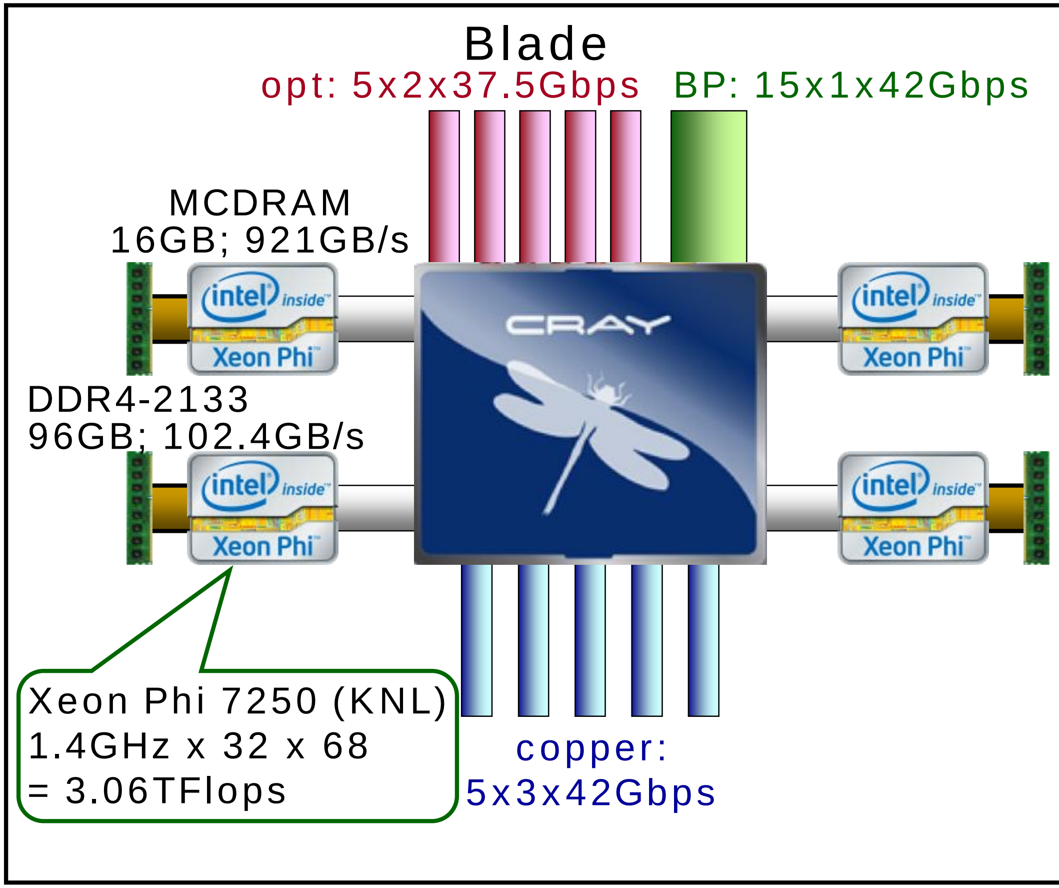
Apology

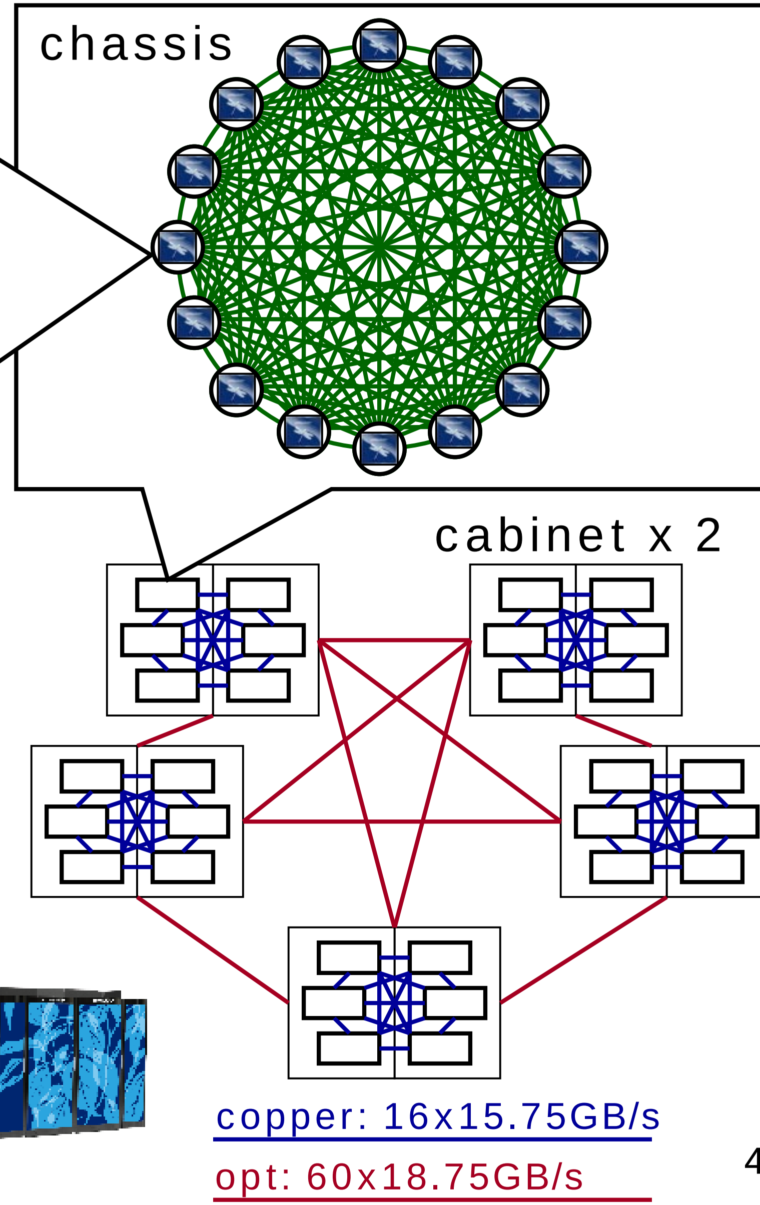
- **I tried to make this talk looking like a keynote by showing a big picture of SIMD-aware compilation for Xeon Phi and its successors.**
- **However unfortunately, I took a wrong way to prepare this talk, examination of Xeon Phi codes generated by representative compilers, and found so many funny things that I cannot resist reporting them in this talk.**
- **Therefore, I'm so sorry that this talk has many nerd (or "otaku" in Japanese) issues about compilers targeting AVX-512, which however I still hope are meaningful not only for compiler people but also for HPC people working on Xeon Phi in general.**

Introduction

- **Xeon Phi's key technologies are;**
  - **high per-core DPFP performance of 32FLOP/cycle achieved by dual-issue 512-bit FMA;**
  - **68 (or 64) x86 cores for up to 272 (or 256) threads;**
  - **high bandwidth (≈500GB/s) MCDRAM;**
  - **and ...**
- **Per-core performance heavily depends on;**
  - **vectorizability of your innermost loops; and**
  - **ability of your compiler;**
    - **to recognize your loops as vectorizable; and**
    - **to generate good code exploiting AVX-512's advanced features (mask, gather/scatter, conflict detection, ...).**
- **Let's see the ability of a few compilers.**

Supercomputer with Xeon Phi in Kyoto





- 68C x 1,800 = 122,400C
- 5.48PFlops
- 28+169=197TB
- 15.5TB/s

How to See the Ability

- **Two programs**
  - **A kind of simple benchmark of `c[i]=a[i]+b[i]` and its variants with index arrays.**
  - **A particle-in-cell (PIC) simulation code having three fairly complicated vectorizable loops.**
- **Programs are written;**
  - **in C99 so that arrays/pointers in loops are `restrict`-ed and multi-dimensional arrays are variable-size in lower dimensions.**
  - **without any intrinsic functions, compiler-specific directives, or `omp simd` pragmas.**
- **and compiled by;**
  - **icc 17.0.3/18.0.0, craycc 8.6.3 and gcc 7.2.0.**

Why without Directives?

- **We accept OpenMP's directive-assisted parallelization because;**
  - **parallelization has too many alternatives to choose the best automatically;**
  - **even for a particular method, examining its applicability is extremely tough; and**
  - **attaching directives is considered as part of parallel programming rather than tuning.**
- **SIMD-vectorization has a different story;**
  - **auto-vectorization is much easier than auto-parallelization; and**
  - **attaching directives to many vectorizable loops is simply boring and harmful for code maintenance.**

Vector Addition: Overview

- **Is `for(i=0;i<n;i++)`*body*`;` vectorized?**

```
double *restrict a, *restrict b, *restrict c;
int *restrict xa, *restrict xb, *restrict xc;
```

body	icc17	icc18	craycc	gcc
`c[i]=a[i]+b[i]`	Yes	Yes	Yes	Yes
`c[i]=a[xa[i]]+b[xb[i]]`	Yes	Yes	Yes	Yes
`c[xc[i]]= a[xa[i]]+b[xb[i]]`	Yes	No	Yes	No
`a[i]+=b[i]`	Yes	Yes	Yes	Yes
`a[i]+=b[xb[i]]`	Yes	Yes	Yes	Yes
`a[xa[i]]+=b[xb[i]]`	Yes	No	No	No

why degrade halving performance?

Vector Addition: Loop Structure (1/2)

- **Common conceptual structure**

```
for(i=0;(long)(c+i)&0x3f;i++) c[i]=a[i]+b[i];
                                               //peeling
for(;i<(n/16)*16;i++) c[i]=a[i]+b[i]; //main
for(;i<n;i++) c[i]=a[i]+b[i];         //remainder
```

- **Compiler-specific features & #instructions**
  - **Average of all possibilities when icc's main loop for `c[i]=a[i]+b[i]` iterates N-times.**
  - **K=3 is #-of kernel instructions in the main body.**

	peeling	main	remainder
icc	vectorized 0.9K+43.1=45.8	2way unroll (2K+3)N=9N	vectorized 1.4K+30.3=34.4
craycc	no 14	2way unroll (2K+10)N=16N	(8+4+2+1)-way 2.9K+18.4=27.0
gcc	expanded scalar (seq of body + if) 3.5K+47.3=57.8	not unrolled (2K+8)N=14N	expanded scalar (seq of body + if) 4K+25.6=37.6

Vector Addition: Loop Structure (2/2)

- **Vectorizing peeling & remainder loops**
  - **Exploits Opmask (`k0-7`) being a new feature of AVX-512 to vectorize very short loops, up to 7 (peeling) or 15 (remainder).**
  - **Fundamentally good idea and effective especially when K is large while N is not so large.**
  - **However, the constant overhead of 30 or so instructions mainly for masking is not negligible especially when N is very small, e.g. 1 or 2, or even 0, in SpMV with a CRS matrix.**
  - **The overhead can be reduced by, e.g.;**
    - **eliminating redundant loop-control instructions for a loop iterating only once.**
    - **introducing new instructions to produce Opmask value from the loop count (like ARM-SVE's `whilelt`).**

Vector Addition: Main Body (1/5)

- **icc17=icc18**

c[i]=a[i]+b[i]	a[i]+=b[i]
vmovups a[i] vmovups a[i+8] vaddpd b[i] vmovupd c[i]= vaddpd b[i+8] vmovupd c[i+8]= addq i+=16 cmpq i<n jb if(i<n)goto	vmovups a[i] vmovups a[i+8] vaddpd b[i] vmovupd a[i]= vaddpd b[i+8] vmovupd a[i+8]= addq i+=8 cmpq i<n jb if(i<n)goto

craycc	gcc
■ Has prefetcht0 for {abc}[i+{80,88}].	■ Not unrolled.
■ Has subq/leaq to increment vmovupd's index (=i*8) because it is not scaled.	

Vector Addition: Main Body (2/5)

- **icc17=icc18**

c[i]=a[xa[i]]+b[xb[i]]		a[i]+=b[xb[i]]	
vmovdqu	xa[i]	vmovdqu	xb[i]
kxnorw	k1=11...11	vpxord	bb=0
vmovdqu	xb[i]	kxnorw	k1=11...11
vpxord	aa=0	vmovups	aa=a[i]
vpxord	bb=0	vgatherdpd	bb=b[]{k1}
kxnorw	k2=11...11	vaddpd	aa+bb
vgatherdpd	aa=a[]{k1}	vmovupd	a[i]=aa+bb
vgatherdpd	bb=b[]{k2}	addq	i+=8
vaddpd	aa+bb	cmpq	i<n
vmovupd	c[i]=aa++bb	jb	if(i<n)goto
addq	i+=8		
cmpq	i<n		
jb	if(i<n)goto		

- **Masking with 11....11 is necessary, but zero-clear (=craycc) of vgatherdpd's destination should be redundant.**
- **craycc & gcc perform 2-way unrolling.**

Vector Addition: Main Body (3/5)

- **Why `ki=11...11` and masking necessary?**
  - **`vgatherdpd` clears `ki` for completed elements so that it can be re-executed when an element causes memory access fault without accessing completed elements repeatedly.**
- **Really necessary?**
  - **`vmovupd` may cross a page boundary and seems to be re-executed as a whole when one of two pages causes memory access fault.**
  - **ARM-SVE's gather (and scatter) does not have such a feature.**
  - **But unfortunately, we cannot make `vgatherdpd` unmasked because it raises #UD exception (sigh).**

Vector Addition: Main Body (4/5)

- **icc17**

```
c[xc[i]]=a[xa[i]]+b[xb[i]]
```

```
vmovdqu      xa[i]
kxnorw       k1=11...11
vmovdqu      xb[i]
vpxord       aa=0
vpxord       bb=0
kxnorw       k2=11...11
vmovdqu      xc[i]
addq         i+=8
kxnorw       k3=11...11
vgatherdpd   aa=a[]{k1}
vgatherdpd   bb=b[]{k2}
vaddpd       aa+bb
vscatterdpd  c[]=aa+bb{k3}
cmpq         i<n
jb           if(i<n)goto
```

works well even when xc[i..i+7] has duplications.

- **craycc performs 2-way unrolling.**

Vector Addition: Main Body (5/5)

- icc17 for a[xa[i]]+=b[xb[i]]

```
L0:vmovdqu       xb[i]
   vpxord        bb=0
   kmovw         k2=11...11
   vpxord        aa=0
   vmovdqu       xb[i]
   kmovw         k3=11...11
   vgatherdpd    bb=b[]{k2}
   vmovdqu       xa[i]
   vpconflictd   c=conf(xa[i])
   vgatherdpd    aa=a[]{k3}
   vpmovzxdq     discard_upper(c)
   vptestmq      k0<j>=(c[j]!=0)
   vaddpd        ab=aa+bb
   kmovw         g=k0
   testl         g==0
   je            if(!g)goto L2
   vpbroadcastmb2q for c[j]!=0
```

```
   vpbroadcastq n[j]=0x3f
   vplzcntq     m[j]=lz(c[j])
   vptestmq     k0<j>=(c[j]!=0)
   vpsubq       n[j]-=m[j]
   kmovw        g=k0
L1:kmovw        k2=g
   vpbroadcastmb2q d[j]=k2
   vpermpd      ab[j]=ab[n[j]]
   vaddpd       ab+=aa{k2}
   vptestmq     k0<j>=(c[j]&d[j])
   kmovw        g=k0
   testl        g==0
   jne          if(g)goto L1
L2:addq         i+=8
   kmovw        k2=11...11
   vscatterdpd  a[]=ab{k2}
   cmpq         i<n
   jb           if(i<n)goto L0
```

- **Complicated code** for the case xa[i..i+7] has duplications, but reasonably efficient if not, and seems better than serial-if-duplicated in most duplicated cases.

Vector Addition: `restrict` Qualification (1/2)

- **`restrict` qualification of RHS arrays ensure that they are not modified by the assignment of LHS arrays (whose mutual conflicts are also ensured from happening by `restrict`-ing them).**
- **Therefore without `restrict`-ion we cannot expect, in general, that a loop is vectorized even when arrays are actually conflict-free.**
- **However, icc and craycc dare to vectorize non-`restrict`-ed `c[i]=a[i]+b[i]` (and `a[i]+=b[i]`) with an inspector to check c−8<a,b<c and a serial loop for the case this condition holds.**
  - **Personally I don't love this officious vectorization because it could make programmers overestimating vectorization capability.**
  - **Loops with indirection are not vectorized because inspection is virtually impossible.**

Vector Addition: `restrict` Qualification (2/2)

- **Modification-free nature of RHS arrays may be guaranteed by another more intuitive qualification, const for array elements (not for the pointer), but is this sufficient for your compiler?**

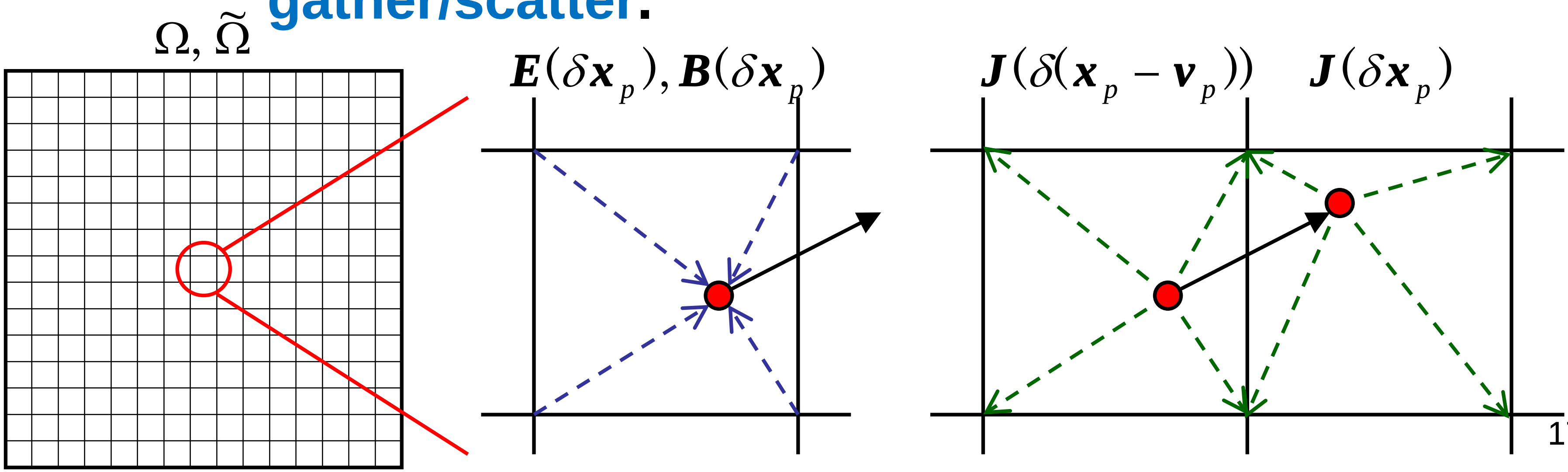
body	icc17	icc18	craycc	gcc
`c[i]=a[i]+b[i]`	Yes/Yes	Yes/Yes	Yes/Yes	Yes/Yes
`c[i]=a[xa[i]]+b[xb[i]]`	Yes/Yes	Yes/Yes	Yes/No	Yes/No
`c[xc[i]]=a[xa[i]]+b[xb[i]]`	Yes/Yes	No/No	Yes/No	No/No
`a[i]+=b[i]`	Yes/Yes	Yes/Yes	Yes/Yes	Yes/Yes
`a[i]+=b[xb[i]]`	Yes/Yes	Yes/Yes	Yes/No	Yes/No
`a[xa[i]]+=b[xb[i]]`	Yes/Yes	No/No	No/No	No/No

  - **Assuring correctness of `const` is easier than `restrict` for both of programmers and compilers.**
  - **In theory, `restrict` qualification of LHS arrays is not necessary because no other arrays appear in LHS.**
  - **However even icc needs `restrict` for LHS arrays, or generates codes for the case without `restrict` at all.**

PIC Code: Overview (1/2)

- **For each *p* at x_p in a cell whose vertices are at δx_p ;**
  - **Update v_p by Lorentz force determined by *E* and *B* at δx_p, and then update x_p by v_p.**
  - **Add the contribution of *p*'s motion to *J* at δx_p.**

  ➔ **In a naive implementation, `E[][][]`, `B[][][]`, `J[][][]` are accessed by $\lfloor x_p \rfloor + \{0,1\}^3$ with gather/scatter.**



PIC Code: Overview (2/2)

- **Let each cell *c* have the set (bin) of all particles in it.**
- **Scalarize *E/B/J* accessed by all *p* in *c*.**

```
for(c in cells){
  {sE}=Earound(c); {sB}=Baround(c);
  for(p in c) v[p]+=lorentz(p,{sE},{sB});
  {sJ}=0;
  for(p in c)
   {{sJ}+=scatter(p); x[p]+=v[p];}
  Jaround(c)+={sJ};
  for(p in c) migrate(p);
}
for(c in cells){
  {sJ}=0;  for(p in c) {sJ}+=scatter(p);
  Jaround(c)+={sJ};
}
```

Since x[] and v[] are simple SOA-type arrays, vectorized well without gather/ scatter of E/B/J.

PIC Code: How Complicated

- **Push-loop for Lorentz acceleration has;**
  - **51 (!!) loop-invariant scalar variables for *E* (24), *B* (24) and the base coordinate of *c* (3).**
  - **149 DP-FLOPs, including a division, for interpolation of *E*/*B*, cross product in Lorentz force calculation, etc.**
- **Two scatter-loops commonly have;**
  - **12 scalar variables to which *J*'s components are accumulated, and 6 loop-invariants for the base coordinate of *c*.**
  - **73 or 66 DP-FLOPs, including three conditional expressions, for extrapolation of the contribution of particle motion to *J*'s components, etc.**

PIC Code: Vectorized?

body	icc17	icc18	craycc	gcc
push	Yes	Yes	Yes	No
scatter-1	Yes	Yes	Yes	No
scatter-2	Yes	Yes	Yes	No

- **Codes generated by icc17 and icc18 are virtually equivalent.**
- **In icc's code, remainder part of all three loops are vectorized, as well as peeling part of push and scatter-2 (while scatter-1 does not have peeling part).**
- **In craycc's code, no loops have peeling part, and their remainder parts are serial.**

PIC Code: Vector Register Allocation

- **For push-loop, icc manages to allocate 16 loop-invariants out of 51 and 2 constants to vector registers, while only 14 registers are used for local/temporary variables.**
- **Even with this good allocation, 35 loop-invariants (and a constant) are kept in memory in fully expanded form (i.e., one variable consumes 64B).**
  - **64B×35=2240B is not small and consumes 6.8% of 32KB L1-Dcache.**
  - **By exploiting *m64bcst* feature, this consumption can be reduced to 280B or 0.85% of L1D.**
  - **Spilled constant is loaded by `vbroadcastsd`.**
- **For two scatter-loops, icc does almost perfect game.**
  - **One constant of scatter-1 is spilled, while three array elements are loaded twice to reduce register consumption.**

PIC Code: Conditionals

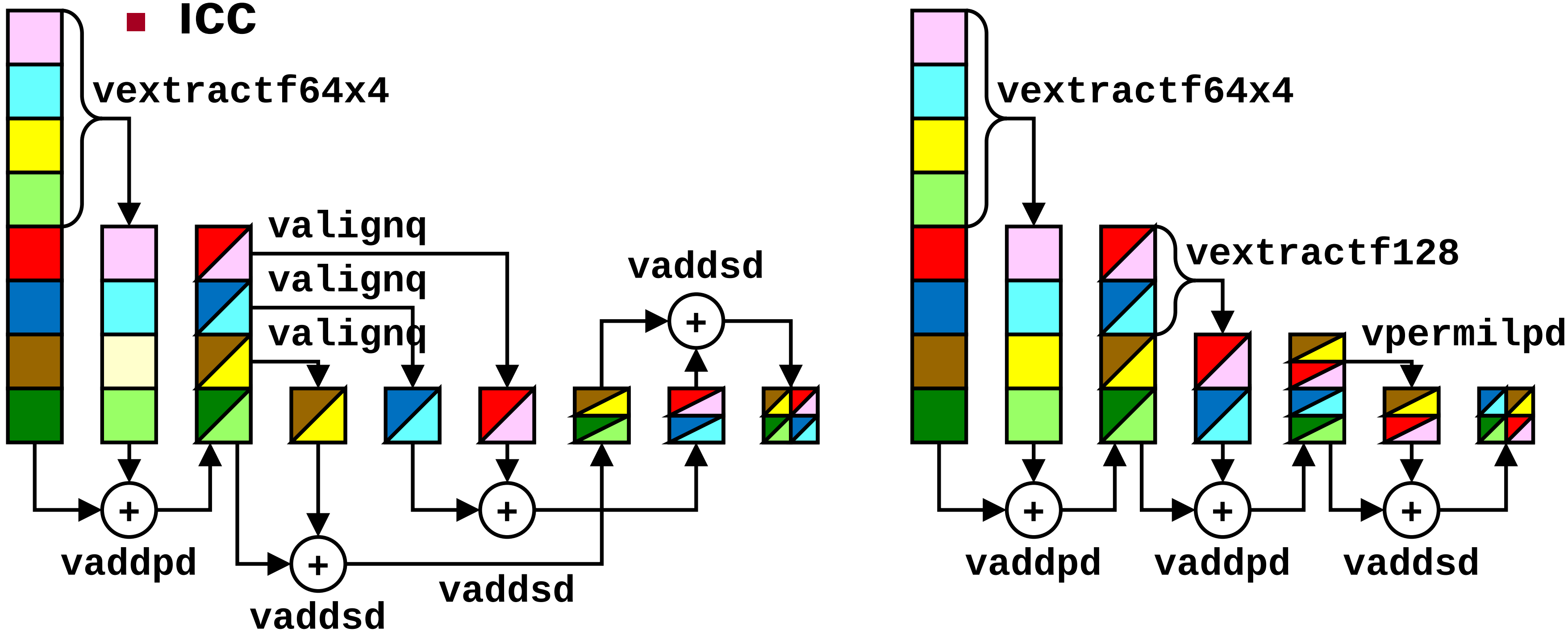
- **Two scatter-loops commonly have;**

```
xr=(x0==x1)?(px0+px1)*0.5:((x0<x1)?x1:x0);
```

- **This conditional expression does not inhibit vectorization in both of icc and craycc;**
  - **Both compilers exploit Opmask.**
  - **icc is a little bit cleverer because it makes `vmulpd` for `(px0+px1)*0.5` masked to overwrite the result of `fmax(x0,x1)`, rather than choosing them by masked `vmovapd`.**
- **However, we cannot expect that loops with any conditionals are vectorized.**
  - **e.g., `for() c[i]=a[i]==0.0?f(a[i],b[i]):a[i]+b[i];` is not vectorized.**
  - **Partial vectorization for the case `a[i..i+7]!=0` seems to be future work (or needs some directive to force vectorization).**

PIC Code: Reductions

- **Summing up 8 partial sums**
  - **icc**



  - **icc's code has two more instructions but its critical path is shorter, by one instruction of moving vector elements.**
  - **Seems efficient even in short vector cases (e.g., dot product for CRS-SpMV).**

PIC Code: Division

- **Push-loop has q=2.0/d**

➔ `(1/d')=vrcp28pd(d);`
`(1/d)=2*(1/d')-d*(1/d')*(1/d');`

  - **icc**

```
(1/d)=(1/d')*(1-d*(1/d'))+(1/d');
q=2*(1/d);
if ((1/d)==NAN) q=vdivid(2,d);
```

Is this exception handling necessary?

  - **craycc**

```
temp=2-d*(1/d');(2/d')=(1/d')+(1/d');
q=temp*(2/d');
```

Optimization(?) for numerator=2.
In general, it will be;
`(num/d')=num*(1/d')`

PIC Code: Peep Holes

- **icc aggressively apply compile-time evaluation of arithmetic expressions.**
  - **Good example**

    source:
    ```
    c=a*b; e=c-d; g=a-c; //a is dead here
    ```
    object:
    ```
    e=a; e=e*b-d; g=a-a*b; //g uses a's reg
    ```
  - **Bad examples**

    source:
    ```
    c=a*b; d=a-c; e=b-c; g+=c*f;
                         //a and b are alive
    ```
    object:
    ```
    c=a*b; d=a; d=a-d*b; e=a; e=b-e*b;
    g+=c*f;
    ```
    source:
    ```
    b=a-x[i]; /*b is used*/ c=(a+b)*0.5;
    ```
    object:
    ```
    b=a-x[i]; /*b is used and dead*/
    c=2*a-x[i]; c*=0.5;
    ```

Closing Remarks

- **Compilers for Xeon Phi (AVX-512), especially icc, generate reasonably efficient codes from C programs free from directives or intrinsics.**
- **However, there is still some room of improvement especially in complicated loop bodies and outside main bodies.**
  - **Outside code has become important as the effective loop trip count has been halved or quartered.**
- **(Micro-)Architectural support is still very welcome.**
  - **Better exception interface of gather/scatter.**
  - **Efficient way to have Opmask for peeling/remainder loops.**
  - **Loop-count-base branch prediction for relatively short loops (e.g. n=10 or so).**
  - **...**