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

Blade
- opt: 5x2x37.5Gbps
- BP: 15x1x42Gbps

MCDRAM
- 16GB; 921GB/s

DDR4-2133
- 96GB; 102.4GB/s

Xeon Phi 7250 (KNL)
- 1.4GHz x 32 x 68
- = 3.06TFlops

Cabinet x 2
- copper: 16x15.75GB/s
- opt: 60x18.75GB/s

- 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 restricted 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?

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

<table>
<thead>
<tr>
<th>body</th>
<th>icc17</th>
<th>icc18</th>
<th>craycc</th>
<th>gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>c[i]=a[i]+b[i]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>c[i]=a[xa[i]]+b[xb[i]]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>c[xc[i]]=</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[xa[i]]+b[xb[i]]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[i]+=b[i]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>a[i]+=b[xb[i]]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>a[xa[i]]+=b[xb[i]]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**why degrade halving performance?**
**Vector Addition:**

**Loop Structure (1/2)**

- **Common conceptual structure**
  
  ```c
  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.

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

- \texttt{icc17=icc18}

\begin{tabular}{|l|l|}
\hline
\texttt{c[i]=a[i]+b[i]} & \texttt{a[i]+=b[i]} \\
\hline
\texttt{vmovups a[i]} & \texttt{vmovups a[i]} \\
\texttt{vmovups a[i+8]} & \texttt{vmovups a[i+8]} \\
\texttt{vaddpd b[i]} & \texttt{vaddpd b[i]} \\
\texttt{vmovupd c[i]=} & \texttt{vmovupd a[i]=} \\
\texttt{vaddpd b[i+8]} & \texttt{vaddpd b[i+8]} \\
\texttt{vmovupd c[i+8]=} & \texttt{vmovupd a[i+8]=} \\
\texttt{addq i+=16} & \texttt{addq i+=8} \\
\texttt{cmpq i<n} & \texttt{cmpq i<n} \\
\texttt{jb if(i<n)goto} & \texttt{jb if(i<n)goto} \\
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
\texttt{craycc} \\
\hline
- Has \texttt{prefetcht0} for \texttt{\{abc\}[i+\{80,88\}].} \\
\hline
\texttt{gcc} \\
\hline
- Not unrolled. \\
\texttt{Has \texttt{subq/leaq} to increment \texttt{vmovupd}'s index (=i*8) because it is not scaled.} \\
\hline
\end{tabular}
**Vector Addition:**  
*Main Body (2/5)*

- **icc17=icc18**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(c[i] = a[xa[i]] + b[xb[i]])</td>
<td>(a[i] += b[xb[i]])</td>
</tr>
<tr>
<td><code>vmovdqu</code></td>
<td><code>xa[i]</code></td>
</tr>
<tr>
<td><code>kxnorw</code></td>
<td><code>k1=11...11</code></td>
</tr>
<tr>
<td><code>vmovdqu</code></td>
<td><code>xb[i]</code></td>
</tr>
<tr>
<td><code>vpxord</code></td>
<td><code>aa=0</code></td>
</tr>
<tr>
<td><code>vpxord</code></td>
<td><code>bb=0</code></td>
</tr>
<tr>
<td><code>kxnorw</code></td>
<td><code>k2=11...11</code></td>
</tr>
<tr>
<td><code>vgatherdpd</code></td>
<td><code>aa=a[]{k1}</code></td>
</tr>
<tr>
<td><code>vgatherdpd</code></td>
<td><code>bb=b[]{k2}</code></td>
</tr>
<tr>
<td><code>vaddpd</code></td>
<td><code>aa+bb</code></td>
</tr>
<tr>
<td><code>vmovupd</code></td>
<td><code>c[i]=aa+bb</code></td>
</tr>
<tr>
<td><code>addq</code></td>
<td><code>i+=8</code></td>
</tr>
<tr>
<td><code>cmpq</code></td>
<td><code>i&lt;n</code></td>
</tr>
<tr>
<td><code>jb</code></td>
<td><code>if(i&lt;n)goto</code></td>
</tr>
</tbody>
</table>

- **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 \( ki=11\ldots11 \) and masking necessary?
  - \( \text{vgatherdpd} \) 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?
  - \( \text{vmovupd} \) 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 \( \text{vgatherdpd} \) unmasked because it raises \#UD exception (sigh).
### Vector Addition: Main Body (4/5)

#### icc17

<table>
<thead>
<tr>
<th>Operation</th>
<th>Data Access</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vmovdqu</code></td>
<td><code>xa[i]</code></td>
</tr>
<tr>
<td><code>kxnorw</code></td>
<td><code>k1=11...11</code></td>
</tr>
<tr>
<td><code>vmovdqu</code></td>
<td><code>xb[i]</code></td>
</tr>
<tr>
<td><code>vpxord</code></td>
<td><code>aa=0</code></td>
</tr>
<tr>
<td><code>vpxord</code></td>
<td><code>bb=0</code></td>
</tr>
<tr>
<td><code>kxnorw</code></td>
<td><code>k2=11...11</code></td>
</tr>
<tr>
<td><code>vmovdqu</code></td>
<td><code>xc[i]</code></td>
</tr>
<tr>
<td><code>addq</code></td>
<td><code>i+=8</code></td>
</tr>
<tr>
<td><code>kxnorw</code></td>
<td><code>k3=11...11</code></td>
</tr>
<tr>
<td><code>vgatherdpd</code></td>
<td><code>aa=a[]{k1}</code></td>
</tr>
<tr>
<td><code>vgatherdpd</code></td>
<td><code>bb=b[]{k2}</code></td>
</tr>
<tr>
<td><code>vaddpd</code></td>
<td><code>aa+bb</code></td>
</tr>
<tr>
<td><code>vscatterdpd</code></td>
<td><code>c[]=aa+bb{k3}</code></td>
</tr>
<tr>
<td><code>cmpq</code></td>
<td><code>i&lt;n</code></td>
</tr>
<tr>
<td><code>jb</code></td>
<td><code>if(i&lt;n)goto</code></td>
</tr>
</tbody>
</table>

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

- craycc performs 2-way unrolling.
Vector Addition: Main Body (5/5)

- **icc17 for** \( a[xa[i]] + b[xb[i]] \)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0: movdqu</td>
<td>( xb[i] )</td>
</tr>
<tr>
<td>vpxord</td>
<td>( bb=0 )</td>
</tr>
<tr>
<td>kmovw</td>
<td>( k2=11...11 )</td>
</tr>
<tr>
<td>vpxord</td>
<td>( aa=0 )</td>
</tr>
<tr>
<td>movdqu</td>
<td>( xb[i] )</td>
</tr>
<tr>
<td>kmovw</td>
<td>( k3=11...11 )</td>
</tr>
<tr>
<td>vgatherdpd</td>
<td>( bb=b[]{k2} )</td>
</tr>
<tr>
<td>movdqu</td>
<td>( xa[i] )</td>
</tr>
<tr>
<td>vpxconflictd</td>
<td>( c=conf(xa[i]) )</td>
</tr>
<tr>
<td>vgatherdpd</td>
<td>( aa=a[]{k3} )</td>
</tr>
<tr>
<td>vpmovzxdq</td>
<td>( discard_upper(c) )</td>
</tr>
<tr>
<td>vptestmq</td>
<td>( k0&lt;j&gt;=(c[j]!=0) )</td>
</tr>
<tr>
<td>vaddpd</td>
<td>( ab=aa+bb )</td>
</tr>
<tr>
<td>kmovw</td>
<td>( g=k0 )</td>
</tr>
<tr>
<td>testl</td>
<td>( g==0 )</td>
</tr>
<tr>
<td>je</td>
<td>( i+=8 )</td>
</tr>
<tr>
<td>vpbroadcastmb2q</td>
<td>( d[j]=k2 )</td>
</tr>
<tr>
<td>L1: kmovw</td>
<td>( k2=g )</td>
</tr>
<tr>
<td>vpbroadcastmb2q</td>
<td>( d[j]=k2 )</td>
</tr>
<tr>
<td>vpermpd</td>
<td>( ab[j]=ab[n[j]] )</td>
</tr>
<tr>
<td>vaddpd</td>
<td>( ab+=aa{k2} )</td>
</tr>
<tr>
<td>vptestmq</td>
<td>( k0&lt;j&gt;=(c[j]&amp;d[j]) )</td>
</tr>
<tr>
<td>kmovw</td>
<td>( g=k0 )</td>
</tr>
<tr>
<td>testl</td>
<td>( g==0 )</td>
</tr>
<tr>
<td>jne</td>
<td>( if(g)goto L1 )</td>
</tr>
<tr>
<td>L2: addq</td>
<td>( i+=8 )</td>
</tr>
<tr>
<td>kmovw</td>
<td>( k2=11...11 )</td>
</tr>
<tr>
<td>vscatterdpd</td>
<td>( a[]=ab{k2} )</td>
</tr>
<tr>
<td>cmpq</td>
<td>( i&lt;n )</td>
</tr>
<tr>
<td>jb</td>
<td>( if(i&lt;n)goto L0 )</td>
</tr>
</tbody>
</table>

- **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.
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?

  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.

<table>
<thead>
<tr>
<th>body</th>
<th>icc17</th>
<th>icc18</th>
<th>craycc</th>
<th>gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c[i]=a[i]+b[i]</code></td>
<td>Yes/Yes</td>
<td>Yes/Yes</td>
<td>Yes/Yes</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td><code>c[i]=a[xa[i]]+b[xb[i]]</code></td>
<td>Yes/Yes</td>
<td>Yes/Yes</td>
<td>Yes/No</td>
<td>Yes/No</td>
</tr>
<tr>
<td><code>c[xc[i]]=a[xa[i]]+b[xb[i]]</code></td>
<td>Yes/Yes</td>
<td>No/No</td>
<td>Yes/No</td>
<td>No/No</td>
</tr>
<tr>
<td><code>a[i]+=b[i]</code></td>
<td>Yes/Yes</td>
<td>Yes/Yes</td>
<td>Yes/Yes</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td><code>a[i]+=b[xb[i]]</code></td>
<td>Yes/Yes</td>
<td>Yes/Yes</td>
<td>Yes/No</td>
<td>Yes/No</td>
</tr>
<tr>
<td><code>a[xa[i]]+=b[xb[i]]</code></td>
<td>Yes/Yes</td>
<td>No/No</td>
<td>No/No</td>
<td>No/No</td>
</tr>
</tbody>
</table>
For each \( p \) at \( x_p \) in a cell whose vertices are at \( \delta x_p \):

- Update \( v_p \) by Lorentz force determined by \( E \) and \( B \) at \( \delta x_p \), and then update \( x_p \) by \( v_p \).
- Add the contribution of \( p \)'s motion to \( J \) at \( \delta x_p \).

In a naive implementation, \( E[[]][[]][] \), \( B[[]][[]][] \), \( J[[]][[]][] \) are accessed by \( \lfloor x_p \rfloor +\{0,1\}^3 \) with gather/scatter.
Let each cell $c$ have the set (bin) of all particles in it.

**Scalarize $E/B/J$ accessed by all $p$ in $c$.**

```python
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);
}
```

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

```python
for(c in cells){
    {sJ}=0; for(p in c) {sJ}+=scatter(p);
    Jaround(c)+={sJ};
}
```
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?**

<table>
<thead>
<tr>
<th></th>
<th>icc17</th>
<th>icc18</th>
<th>craycc</th>
<th>gcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>scatter-1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>scatter-2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

- 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.
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).

- $64B \times 35 = 2240B$ is not small and consumes 6.8% of 32KB L1-Dcache.
- By exploiting \textit{m64bcst} feature, this consumption can be reduced to 280B or 0.85% of L1D.
- Spilled constant is loaded by \textit{vbroadcastsd}.

For two scatter-loops, icc does \textit{almost perfect game}.

- One constant of scatter-1 is spilled, while three array elements are loaded twice to reduce register consumption.
Two scatter-loops commonly have;
\[ \text{xr} = (x0 == x1) \? (px0 + px1) \times 0.5 : ((x0 < x1) \? x1 : x0) ; \]

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) \times 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**
  - `vextractf64x4`
  - `valignq`
  - `vaddsd`
  - `vaddpd`

- **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 $q = 2.0/d$
  
  $$(1/d') = \text{vrcp28pd}(d);$$
  
  $$(1/d) = 2 \times (1/d') - d \times (1/d') \times (1/d');$$

- icc
  
  $$(1/d) = (1/d') \times (1 - d \times (1/d')) + (1/d');$$

  $q = 2 \times (1/d);$ 
  
  if $((1/d) == \text{NAN})$, $q = \text{vdivid}(2, d);$ 

- craycc
  
  $\text{temp} = 2 - d \times (1/d'); (2/d') = (1/d') + (1/d');$

  $q = \text{temp} \times (2/d');$

Is this exception handling necessary?

Optimization(?) for numerator=2. In general, it will be;

$$(\text{num}/d') = \text{num} \times (1/d')$$
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;
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).
  - ...
  - ...