Limits of sustained performance and remedies

WSSP 28
9.-10. October 2018

Uwe Küster

Outline

Performance of a Parallel Pipeline 1/3

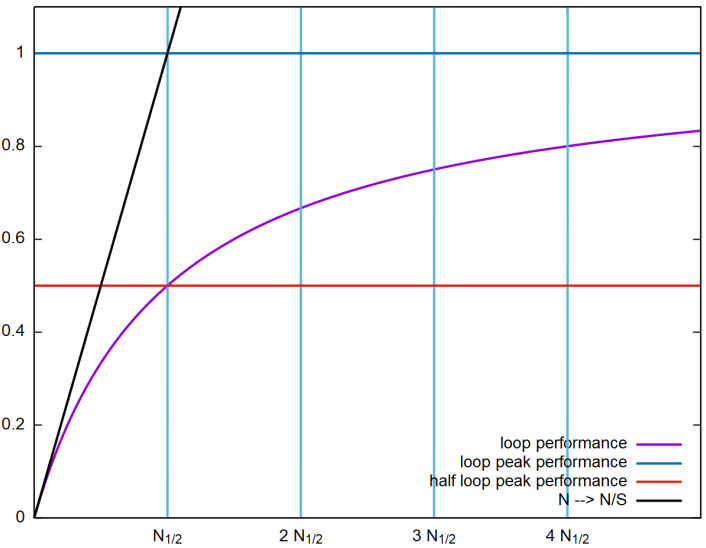
pipelines are realized in many internal mechanisms of modern computers. Assume a parallelizable or vectorizable loop or loop nest of any complexity.

- $\#op$ the number of operations of a single loop iteration
- p is the degree of parallelism (e.g. SIMD parallelism, core parallism).
- Let Δt be the time for a single inner iteration on the hardware.

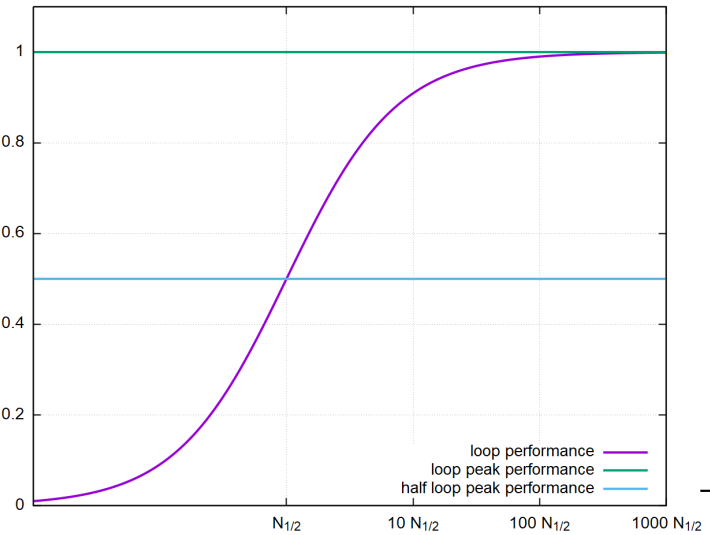
$$\Delta t = \max\left(t_{arith}, t_{BW}, t_{adm}\right) \tag{1}$$

  t_{arith} for arithmetic operations
  t_{BW} to get the data (Small bandwidth increases Δt)
  t_{adm} for inner administration
  All data of the $p-$fold inner (SIMD-) iteration must be present.
- S is the time for the startup respective the length of the pipeline. S is the time for getting the first result. S is related to the latency to get the data from memory or caches or registers. In case of parallel operations as for OpenMP the startup includes the administration time for the parallel setup and ending clauses.
- n is the total number of iterations of the loop or loop nest (not the inner iteration on the hardware). This might be the product of the extents of all nested loop iterations.

Performance of a Parallel Pipeline 2/3



linear loop length



logarithmic loop length

- The performance in dependence of the loop length n with these parameters is

$$Perf\,(n) = \frac{\#op\ n}{S + \frac{n}{p}\ \Delta t} = \frac{\#op}{\frac{S}{n} + \frac{1}{p}\Delta t} \qquad (2)$$

- For large n this approximates the **loop peak performance**

$$Perf\,(n) \overset{n\to\infty}{\longrightarrow} \frac{\#op\ p}{\Delta t} \qquad (3)$$

- Inserting in (2) the special length

$$n_{1/2} = \frac{S\ p}{\Delta t} \qquad (4)$$

we get the loop length $n_{1/2}$ providing **half of the loop peak performance**.

- exceeding $n_{1/2}$ for the loop length is important for sustained performance
- The idea of $n_{1/2}$ is due to R.W. Hockney [a]

[a] R.W. Hockney , Parametrization of computer performance in Parallel Computing, Volume 5, Issues 1–2, July 1987, Pages 97-103

Performance of a Parallel Pipeline: properties of $n_{1/2}$ 3/3

$$n_{1/2} = \frac{S\ p}{\Delta t}$$

- Small $n_{1/2}$ would simplify performance programming
- Decreasing the start up S decreases $n_{1/2}$.
- Increasing the SIMD factor p for constant Δt increases $n_{1/2}$!
- If $\Delta t = \Delta t_{BW}$ is dominated by the data transfer, increasing the system bandwidth would increase (!) $n_{1/2}$ (surely also the performance).
- Their are two ways out of the dilemma (assuming constant frequencies)
  - overlap the startup S by usefull operations, so it will hidden and no more relevant
  - handle nested loops together to get an effective operation count higher than the single loop $n_{1/2}$
- Hardware supported Out of Order Operation facilities are essential for hiding the startup latencies; many instruction substreams have to work at the same time. Some architectures use multithreading (forcing additional explicit parallelization); prefetching is helpful but might introduce additional instructions. Better would be an automatic run time rearrangement of instructions of each stream.

Performance of a Parallel Pipeline: triad on Aurora as example

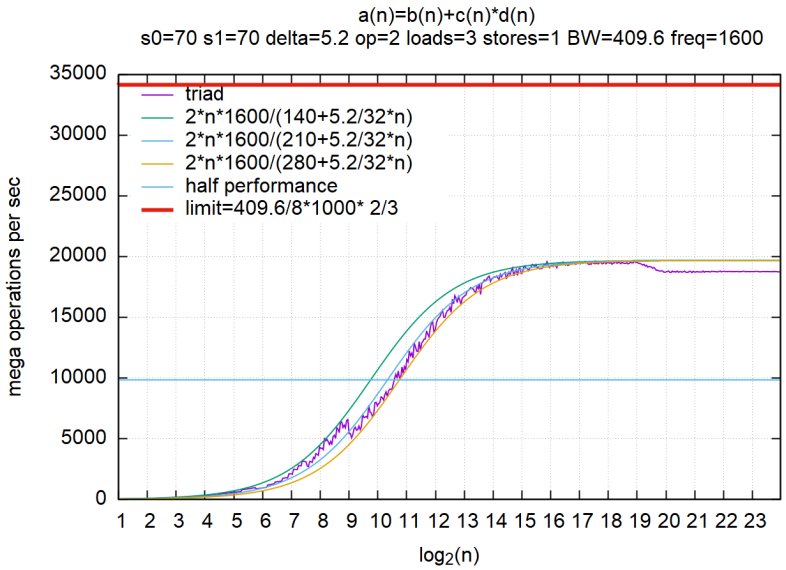


Figure: with cache bandwidth limit

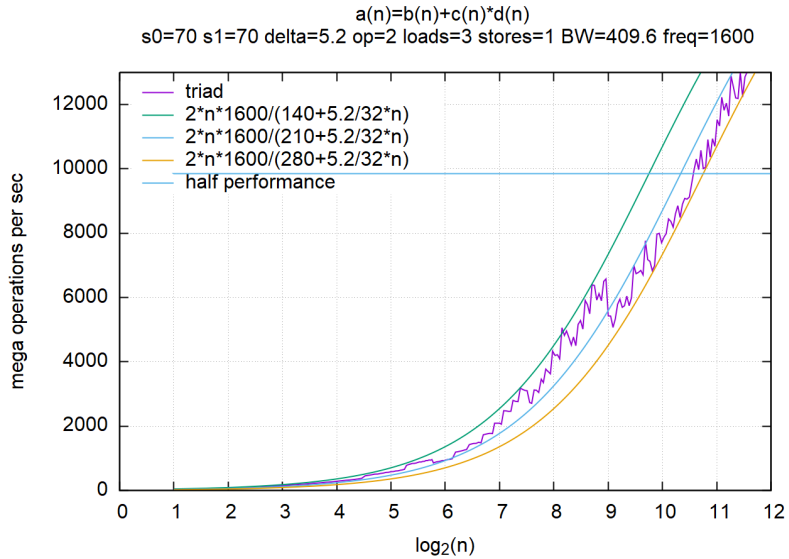


Figure: part for smaller loop length

```
include "begin_of_time_measurement.inc"
b(1)=start_value
do 10 k=1,nu_rep
  do 20 n=1,nmax
    a(n)=b(n) + c(n)*d(n)
  20 continue
  if(a(nmax)>42.) call side_effect(b)
10    continue
end_value=a(nmax)
include "end_of_time_measurement.inc"
```

- $\Delta t = 5.2$ in (2).
- We expected $\Delta t = 3$ generated by the loads with 1 hidden store and 1 hidden Floating-Mult-Add operation.

Performance einer Parallel Pipeline: nested triad as example

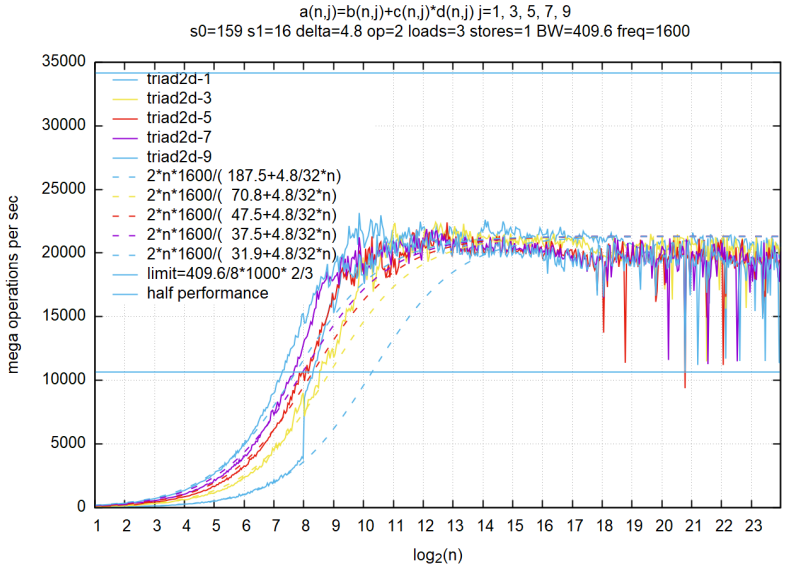


Figure: loop peak performance as measured

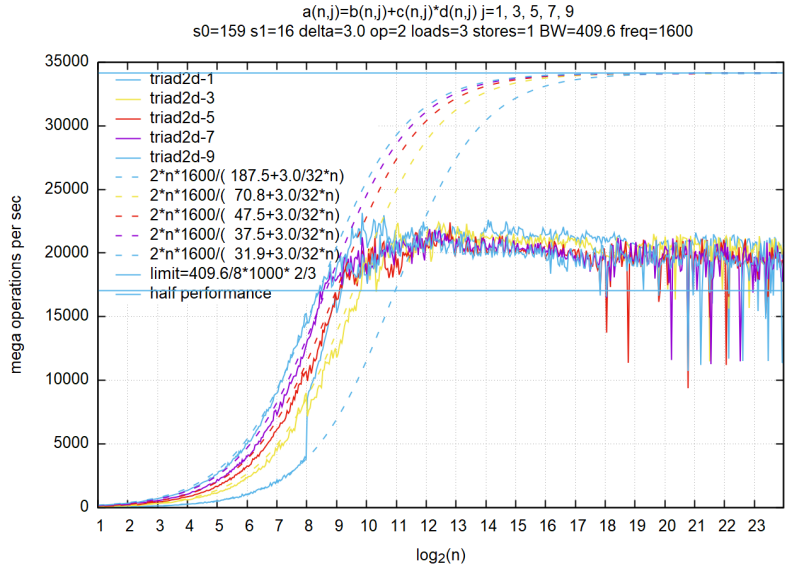


Figure: loop peak performance by cache BW limit

overlapping loop nest:

```
include "begin_of_time_measurement.inc"
b(1,1)=start_value
do 10 k=1,nu_rep
  do j=1,jmax
    do n=1,nmax
      a(n,j)=b(n,j) + c(n,j)*d(n,j)
    enddo
  enddo
  b(1,1:jmax)=a(nmax,1:jmax)
10 continue
end_value=sum(a(nmax,1:jmax))
include "end_of_time_measurement.inc"
```

- cache bandwidth is exhausted for smaller loop length
- $n_{1/2}$ is reduced by overlap but still large
- $\Delta t = 3$ for medium inner loop lenghts
- but no increase for larger loop length
- see performance fluctuations

time consuming kernels in simulation

- Sometimes loops counts are large due to the nature of the kernel:
  - Krylov-space algorithms,
  - large matrices,
  - sparse matrix multiplication with long inner loop (Jagged Diagonal)
- in many cases there are nested loops to be handled:
  - many medium and small sized matrices,
  - handling of pixels and voxels for images,
  - Fast Fourier Transform with varying length
  - structured and unstructured grids in CFD,
  - definition of Finite Elements and assembly
- sometimes long outer loops and short nested inner loops:
  - sparse matrix row ordered matrix vector multiplication,
  - loops over cells or faces for CFD,
  - particle codes,
  - handling high order multidimensional polynoms

How to handle nested loops, all with short lenght, but having many operations in total? Sometimes the iterations are independent, sometimes they include reductions, sometimes the inner loop length depends on the outer.

parametrized vector registers on Aurora

- NEC Aurora has special load and store instructions (VLD2D/VST2D) allowing for parametrized access of data
- This allows for parametrized addressing of array data rectangles in nested loops during load in a single vector register
- vector registers with the identical parametrization can be combined in expressions
- storing from a vector register is using the same mechanism
- different to unrolling!

Addressing in the VLD2D/VST2D instruction 1/2

- Assume

$$VL \in [1, 256] \text{ vector length; here rectangle size} \quad (5)$$
$$os \text{ address offset} \quad (6)$$
$$s1, s2 \text{ address differences ; may be } = 0 \quad (7)$$

- Two addressing functions

$$[0, 15] \ni r1\,(i) = i\%16 \qquad \text{given by bits } [0, 1, 2, 3] \quad (8)$$
$$\left[0, \tfrac{VL-1}{16}\right] \ni r2\,(i) = i/16 \qquad \text{given by bits } [4, 5, 6, 7] \quad (9)$$

- instructions as VLD2D/VST2D allow data to be loaded/stored by the parametrization

$$[0, VL-1] \ni i \longrightarrow os + s1 * r1(i) + s2 * r2(i) \quad (10)$$

- os,$s1$,$s2$ are part of the instruction

Addressing a multidimensional array

- Assuming a Fortran array a with contiguous enumeration of the elements and the dimensions $d1, d2, d3$ (up to 15)

```
real(kind=REAL64),dimension(d1,d2,d3) :: a
```

  then the elements of a are related to the linear memory adressing by

$$
\begin{aligned}
&a\,(m1, m2, m3) \\
&= ValueOfAddress\,(d0 + (m1 - 1) + d1 * (m2 - 1) + d1 * d2 * (m3 - 1))
\end{aligned}
$$

  where $ValueOfAddress\,(j)$ is the hardware based function giving the 8-byte real value at address j.

- In C/C++ the mapping is due to the programmer, but can be (and must be) done in a similar way.
- The compiler must be able to detect this multilinear adressing and its parameters (might be difficult).

Addressing a subarray by VLD2D/VST2D

- Given are offsets $dm1, dm2, dm3$ in the multidimensional array
- Define the offset os and the values $s1, s2$ by

$$\begin{aligned} os &= d0 + (m1 + dm1 - 1) + d1 * (dm2 - 1) + d1 * d2 * (dm3 - 1) \\ s1 &= d1 \\ s2 &= d1 * d2 \end{aligned} \tag{11}$$

- Then we have

$$\begin{aligned} &a\left(m1 + dm1, r1\left(i\right) + dm2, r2\left(i\right) + dm3\right) \\ &\qquad = ValueOfAddress\left(os + d1 * r1\left(i\right) + d1 * d2 * r2\left(i\right)\right) \end{aligned} \tag{12}$$

- (12) shows, how a specific part of the array is accessed by VLD2D/VST2D.
- There is no predefined order of $r1$ and $r2$. The range of $r2$ is limited by $\frac{VL-1}{16}$. The range of $r1$ is in $[0, 15]$ and can be limited by masking.
- the instruction VLD2D/VST2D loads/stores the $16 \times VL/16$ rectangular subarray

  `a(m1+dm1,1+dm1:16+dm1,1+dm2:VL/16+dm2)`

  from memory into the vector register/ from the vector register to memory.

relations between different arrays in an expression, an example

- Given is the following (artificial) loop nest as example

```
real(kind=8) :: a(d1a,d2a,d3a)
real(kind=8) :: b(d1b)
real(kind=8) :: c(d1c)
real(kind=8) :: d(d1d,d2d)
real(kind=8) :: e(d1e)
do m=1,mmax
  do l=1,lmax
    do k=1,kmax
       a(m,l,k)=b(k)*c(l)+d(k,l)+e(m)
    enddo
  enddo
enddo
```

  all dimensions of all arrays may differ; in this case, unrolling is not possible.
- Remark the different order of l, k on the left hand side and k, l on the right hand side.
- Remark the different dimensions of the different arrays.
- There are no dependencies of all operations within the nested loops.

relations between different arrays in an expression in a simple case

- Assume $lmax = 16$ and $kmax = 16$.
- Then the 3-fold nested loop can be replaced by the construct

```
do m=1,mmax
    do i=1,256
       l=r1(i)+1
       k=r2(i)+1
       a(m,l,k)=b(k)*c(l)+d(k,l)+e(m)
    enddo
enddo
```

  which fits directly to the VLD2D mechanism.
- Here the order of $r1$ and $r2$ given in (9) is not relevant. The order is different in a and in d.
- also the 1D arrays b and c are loaded via VLD2D with constant elements in the $r1$ respective $r2$ direction.
- The index m is not relevant, but shows the flexibility of the approach.
- The following pages show the general case with 2D-postambles. Stripmining is more elaborated as in the $1D$-case.
- The mapping of the enumeration of a vector register to multi-indices could be generalized to more than 2 dimensions.

relations between different arrays in an expression

- Assume $lmax > 16$ and $kmax > 16$.
- The inner loops are now replaced by following four nested loops. This is $2D$ stripmining.

```
outer_loop: do m=1,mmax


! case 11
do_l0:  do l0=1,lmax-16,16
  do_k0:  do k0=1,kmax-16,16
    do_i_1: do i=0,255
     l=r1(i)+l0
     k=r2(i)+k0
     a(m,l,k)=b(k)*c(l)+d(k,l)+e(m)
    enddo do_i_1
  enddo do_k0

! case 12
    k1=((kmax-1)/16)*16+1
    VL=(kmax-k1+1)*16
    do_i_2:   do i=0,VL-1
     l=r1(i)+l0
     k=r2(i)+k1
     a(m,l,k)=b(k)*c(l)+d(k,l)+e(m)
    enddo do_i_2
enddo do_l0
```

```
! case 21
    l1=((lmax-1)/16)*16+1
    VL=(lmax-l1+1)*16
do_k0_2:    do k0=1,kmax-16,16
   do_i_3:  do i=0,VL-1
     l=r2(i)+l1   ! different order
     k=r1(i)+k0   ! different order
     a(m,l,k)=b(k)*c(l)+d(k,l)+e(m)
   enddo do_i_3
enddo do_k0_2

! case 22
    k1=((kmax-1)/16)*16+1
    l1=((lmax-1)/16)*16+1
    VL=(lmax-l1+1)*16
    do_i_4:  do i=0,VL-1
       l=r2(i)+l1   ! different order
       k=r1(i)+k1   ! different order
      if(k<= kmax) then ! for r1 term
        a(m,l,k)=b(k)*c(l)+d(k,l)+e(m)
      endif
    enddo do_i_4
enddo outer_loop
```

example 1/5

- let $lmax = 18$ and $kmax = 19$.
- The following example shows the internal addressing by $0 \leq i \leq 256$ in the resulting 4 steps.

example 2/5

case_11

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
18	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
17	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
16	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	–	–	–
15	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	–	–	–
14	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	–	–	–
13	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	–	–	–
12	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	–	–	–
11	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	–	–	–
10	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	–	–	–
9	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	–	–	–
8	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	–	–	–
7	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	–	–	–
6	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	–	–	–
5	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	–	–	–
4	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	–	–	–
3	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	–	–	–
2	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	–	–	–
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	–	–	–

example 3/5

case_12

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
18	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	–	–	–
17	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	–	–	–
16	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
15	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
14	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
13	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
12	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
11	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
10	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
8	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
7	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
5	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

example 4/5

case_21

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
18	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
17	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
16	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	16	32	48
15	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	15	31	47
14	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	14	30	46
13	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	13	29	45
12	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	12	28	44
11	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	11	27	43
10	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	10	26	42
9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	9	25	41
8	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	8	24	40
7	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	7	23	39
6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	6	22	38
5	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	5	21	37
4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	4	20	36
3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	3	19	35
2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	2	18	34
1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	1	17	33

(13)

example 5/5

case_22

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
18	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	2	18	34
17	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	1	17	33
16	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
15	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
14	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
13	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
12	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
11	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
10	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
8	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
7	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
5	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

How to identify the loops of interest

- Clearly we expect the compiler to do the necessary work.
- Deciding to $1D$-vectorize or to $2D$-vectorize and what combination of nested loops to select depends on run time parameters.
- We recommend directives/pragmas in front of the two loops in a loop nest to identify the combination for the $2D$-vectorization, may be depending on these parameters.
- The mechanism could be generalized for vectorization for even more nested loops dimensions.
- What to do with loops with iteration counts dependent on outer loops?

What else?

example from BLAS SGEMV:

```
      DO 100 J = 1,N
          TEMP = ZERO
          DO 90 I = 1,M
              TEMP = TEMP + A(I,J)*X(I)
 90       CONTINUE
          Y(JY) = Y(JY) + ALPHA*TEMP
          JY = JY + INCY
100       CONTINUE
```

- A reduction along **I** is done for each **J**.
- This situation appears in many algorithmic contexts, not only linear algebra, not only linear algebra.
- For this purpose it would be usful to have an instruction summing up partial sums in a vector register.
- This could use similar semantics as VLD2D.

conclusions

- The pipeline model forces long (nested) loops for sustained performance
- overlap of instruction sequences essential for performance
- parametrized vector register allow for better usage of vectorization
- other mechanisms to be expected, e.g. vectorized partial sums
- implementation of simple essential kernels in hardware

Thank you for your attention

Uwe Küster
uwe-kuester[at]gmx.net