Why Vector Computer and Vectorization?

Uwe Küster

ISC Frankfurt 27.6.2018
The Future of High Performance Computing

• Limits in performance growth
  o Technology at physical boundaries (→ production costs per chip increasing)
  o Electrical power consumption depends
    • Strongly on frequency (~ f^q with 2.5 <= q <= 3)
    • Linearly on machine size
  o Processor frequency already stagnating or even decreasing
  o Number of cores per processor increasing on multi-dies
  o Memory bandwidth slowly increasing (but not in relation to peak performance)
  o Memory latencies are not decreasing
  o Peak performance increasing by additional functional units

• And the application performance??
  • Application size is limited (remember numerical complexity O(n^α) with α > 1)
    o Number of processes in relevant applications is not increasing arbitrarily → scaling is limited

• Implications
  o → Adapting architectures
  o → adapted programming
Parallelism

- Assuming fixed frequency → performance is completely dependent on parallelism of different types
  - Software based parallelism
    - Appropriate multi-node interconnect
    - Workflows on clusters, clouds
    - Programmed parallelism on cores
    - Programmed by MPI, PGAS, OpenMP
  - Hardware based parallelism
    - Pipeline parallelism
    - SIMD
    - Compiler supported vectorization
  - Parallelization by hardware
    - Automatically overlapping and interchanging instructions for operations and data transfer at run time (Out Of Order Operation)

- Parallelism needs memory bandwidth
- Parallelism to be found in smaller code segments (→low overhead)
As an example: NEC Aurora Core Architecture

- 8 cores sharing the cache
- Vector Engine cooperating with Vector Host
- Additional shadow registers allow for out of order
- Scalar processor for non vectorizable code parts and initiation of vector instructions
NEC Aurora vector registers

Intel AVX-512
Vector register of 8x8B variables = 512 bit
Coupled with 8 way SIMD FMA-unit

• **Pipelined SIMD** operations (8 x 32)
• 3 FMA-units working **out of order**
• Vector registers can address data by 1D→2D map for load/store
  → vectorization of nested loops

Vector Length = 256e (32e x 8 cycle)
307.2GF = 32Flops/cycle x 2(FMA) x 3 x 1.6GHz
some examples
• Different behaviour with respect to \( j_{\text{max}} \) because of overlap
• Recursion at the end forces the loop to start with no overlap
A more costly loop

Derived type:

```
dtype state_simple_type
type state_simple_type
    real(kind=test_kind) :: rho
    real(kind=test_kind) :: rhov1,rhov2,rhov3
    real(kind=test_kind) :: rhoE
end type state_simple_type
```

Loop:
```
rhoi = 1./state(i)%rho
EE = state(i)%rhoE
L2norm2=state(i)%rhov1**2+state(i)%rhov2**2+state(i)%rhov3**2
pp = gami * (EE - onehalf*L2norm2)
nss = gamma*pp * rhoi**gamma
HH = rhoi * (EE + pp)

simple(i)%rho = log(ss)
simple(i)%rhov1 = rhoi * state(i)%rhov1
simple(i)%rhov2 = rhoi * state(i)%rhov2
simple(i)%rhov3 = rhoi * state(i)%rhov3
simple(i)%rhoE = log(HH)
```

Derived type involves stride
Intrinsic functions

Operations counted as
```
! #:  9 *;  4 +;  1 /;  1 **;  2 log
! CP:  1;  1;  3;  6;  9
timings might not be appropiate!
Row ordered sparse matrix vector multiplication

Row

1
2
3
4
5
6
7
8

pseudo column

<table>
<thead>
<tr>
<th>row</th>
<th>column</th>
<th>column</th>
<th>column</th>
<th>column</th>
<th>column</th>
<th>column</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
<td>value</td>
</tr>
</tbody>
</table>

1 2 3 4 5 6 7 8

!$OMP PARALLEL PRIVATE(n,m,offset)
!$OMP DO
do n=1,nmax
offset=begin(n)
y(n)=0.
do m=1,length(n)
y(n)=y(n)+a(offset+m)*x(index(offset+m))
endo
dndo
!$OMP END DO
!$OMP END PARALLEL

Reduction expensive
Unrolling difficult
Short vector length, poor performance
Simply to define, typical
Used in all packages
Jagged diagonal storage format

Unrolling difficult
Long vector length, good for vectorization
Definition by reordering of row ordered scheme
Cache blocking difficult
Reordering of result vector necessary
Parallelization by blocks of rows

http://www.netlib.org/linalg/html_templates/node95.html
Blocked jagged diagonal

OpenMP with 8 cores
Near peak bandwidth

Unrolling reasonable because one jump needed for the block
Block width can be fixed
Block length can be limited to vector register length
Vectorization: Example of Compilation listing

130: vec( 101): Vectorized loop.
130: err( 504): The number of VLOAD, VSTORE.: 1, 0.
130: err( 505): The number of VGT, VSC.: 0, 0.
131: opt(1019): Feedback of scalar value from one loop pass to another.: TEMP
131: vec( 126): Idiom detected.: ITERATION

Vectorization properties similar to SIMD-vectorization

PROCEDURE NAME: FORTRAN_EXAMPLES_MODULE::HORNER_SCHEME
FORMAT LIST
122: function horner_scheme(a,n,lambda) result(result_value)
123:     integer :: n
124:     real :: a(n)
125:     real :: lambda
126:     real :: result_value,temp
127:     integer :: i
128:
129:     temp=a(n)
130:     do i=n-1,0,-1
131:         temp=a(i)+lambda*temp
132:     enddo
133:     result_value=temp
134:
135: end function horner_scheme

nfort -c -O4 -report-all -fdiag-vector=3 Fortran_examples.f90
Vectorization and C-typical constructs

```c
#include <stddef.h>
typedef struct {  
    float* data;  
    size_t size;  
} vec_t;

void vec_eltwise_product(vec_t* a, vec_t* b, vec_t* c) {
    size_t i;
    for (i = 0 ; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}

void vec_restrict_product(vec_t* a, vec_t* b, vec_t* c) {
    size_t i;
    float* __restrict da=&a->data[0];
    float* __restrict db=&b->data[0];
    float* __restrict dc=&c->data[0];
    int da_size=a->size;
    for(i = 0 ; i < da_size; i++) {
        dc[i]=da[i]+db[i];
    }
}

void vec_pointer_product(float* da, float* db, float* dc,int* da_size) {
    size_t i;
    for(i = 0 ; i < *da_size; i++) {
        *dc =*da+*db;
        dc++;da++;db++;
    }
}
```

ncc -c -O4 -report-all -fdiag-vector=3 my_test.c
NEC Aurora conclusions

- High processor performance
- High memory bandwidth
- Low electrical power
- Loops must be vectorizable
- Work distribution between Vector Engine and Vector Host
Thank you for your interest

• uwe-kuester@gmx.net