High Performance Computing: Tools and Applications

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology

Lecture 9

SIMD vectorization using `#pragma omp simd`

- force compiler to transform loop into a SIMD loop, even if there are data dependencies
- compiler will still generate code for peeling and remainder loop
- compiler will issue a warning (by default) if loop cannot be vectorized
- clauses
  - `safelen(length)`
  - `simdlen(length)`
  - `linear(list)`
  - `aligned(list)`
  - `private(list)`
  - `lastprivate(list)`
  - `reduction(list)`
  - `collapse(n)`
- Note: Intel has its own `#pragma simd` which is almost identical and can be used if you are not using OpenMP (but needs Intel compiler)

Example `simd.c`

```
    double *a0 = (double *) _mm_malloc(n*sizeof(double),64);
    double *a = a0;

#pragma omp simd
    for (i=0; i<n; i++) {
        // __assume_aligned(a,64);
        a[i] = 0.;
    }

#pragma omp simd
    for (i=1; i<=n; i++)
        a[i] = a[i-1] + 1.;

    printf("a[n-1]: %f\n", a[n-1]);

    _mm_free(a0);
```

```
icc -qopt-report=5 -qopt-report-phase=vec \
  -qopt-report-file=stdout -qopenmp simd.c
```

Does this auto-vectorize?

```
void add(double *a, double *b, double *c)
{
    *c = *a + *b;
}

void main()
{
    ...
    for (i=0; i<n; i++)
        add(&a[i], &b[i], &c[i]);
}
```

Does this auto-vectorize?

```
void add(double *a, double *b, double *c)
{
    *c = *a + *b;
}

void main()
{
    ...
    for (i=0; i<n; i++)
        add(&a[i], &b[i], &c[i]);
}
```

Yes, actually, because the function is inlined.

Ordinarily, the loop containing a function call cannot be vectorized.

SIMD-enabled functions (a.k.a. elemental functions)

Tell the compiler to make a vectorized version of the function

```
#pragma omp declare simd
__declspec(noinline) // no inline for this example
void add(double *a, double *b, double *c)
{
    *c = *a + *b;
}

void main()
{
    ...
    for (i=0; i<n; i++)
        add(&a[i], &b[i], &c[i]);
}
```

Thread affinity: `verbose` mode

```
void main()
{
    #pragma omp parallel num_threads(4)
    {
        int threadid = omp_get_thread_num();
        printf("phase 1: %d\n", threadid);
    }

    printf("----------\n");
    fflush(stdout);

    sleep(1);

    #pragma omp parallel num_threads(5)
    {
        int threadid = omp_get_thread_num();
        printf("phase 2: %d\n", threadid);
```

Thread affinity: `verbose` mode

```
joker:~/cse6230/samples-lec10$ KMP_AFFINITY=granularity=fine,verbose,compact ./a.out
OMP: Info #204: KMP_AFFINITY: decoding x2APIC ids.
OMP: Info #202: KMP_AFFINITY: Affinity capable, using global cpuid leaf 11 info
OMP: Info #154: KMP_AFFINITY: Initial OS proc set respected: {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,1
OMP: Info #156: KMP_AFFINITY: 40 available OS procs
OMP: Info #157: KMP_AFFINITY: Uniform topology
OMP: Info #179: KMP_AFFINITY: 2 packages x 10 cores/pkg x 2 threads/core (20 total cores)
OMP: Info #206: KMP_AFFINITY: OS proc to physical thread map:
OMP: Info #171: KMP_AFFINITY: OS proc  0 maps to package 0 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 20 maps to package 0 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc  1 maps to package 0 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 21 maps to package 0 core 1 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc  2 maps to package 0 core 2 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 22 maps to package 0 core 2 thread 1
...
OMP: Info #242: KMP_AFFINITY: pid 36172 thread 0 bound to OS proc set {0}
OMP: Info #242: KMP_AFFINITY: pid 36172 thread 1 bound to OS proc set {20}
OMP: Info #242: KMP_AFFINITY: pid 36172 thread 2 bound to OS proc set {1}
phase 1: 0
phase 1: 1
phase 1: 2
OMP: Info #242: KMP_AFFINITY: pid 36172 thread 3 bound to OS proc set {21}
phase 1: 3
----------
phase 2: 0
phase 2: 1
phase 2: 2
phase 2: 3
OMP: Info #242: KMP_AFFINITY: pid 36172 thread 4 bound to OS proc set {2}
phase 2: 4
```

Thread affinity: `verbose` mode

- Info 171 identifies each OS proc by package, core, thread
- Info 242 shows how each thread id is mapped to OS proc
- If additional threads are created, mapping of new threads is shown (mapping cannot change if new threads are created)

Thread affinity: tests with `bench-dgemm.cc`

```
#include <mkl.h>
#include <stdio.h>
#include <omp.h>

int main() {
    printf("mkl max threads: %d\n", mkl_get_max_threads());
    const int N = 10000; const int Nld = N+64;
    const char tr='N'; const double v=1.0;
    double* A = (double*)_mm_malloc(sizeof(double)*N*Nld, 64);
    double* B = (double*)_mm_malloc(sizeof(double)*N*Nld, 64);
    double* C = (double*)_mm_malloc(sizeof(double)*N*Nld, 64);
    _Cilk_for (int i = 0; i < N*Nld; i++) A[i] = B[i] = C[i] = 0.0f;
    int nIter = 10;
    for(int k = 0; k < nIter; k++)
    {
        double t1 = omp_get_wtime();
        dgemm(&tr, &tr, &N, &N, &N, &v, A, &Nld, B, &Nld, &v, C, &N);
        double t2 = omp_get_wtime();
        double flopsNow = (2.0*N*N*N+1.0*N*N)*1e-9/(t2-t1);
        printf("Iteration %d: %.1f GFLOP/s\n", k+1, flopsNow);
    }
    _mm_free(A); _mm_free(B); _mm_free(C);
}
```

bench-dgemm.cc

```
icpc bench-dgemm.cc -qopenmp -mkl -mmic

KMP_AFFINITY=compact ./a.out
```

bench-dgemm.cc

```
[edmond@joker-mic4 ~]$ ./a.out
num threads: 240
Iteration 1: 216.3 GFLOP/s
Iteration 2: 334.0 GFLOP/s
Iteration 3: 331.6 GFLOP/s

[edmond@joker-mic4 ~]$ KMP_AFFINITY=compact ./a.out
num threads: 240
Iteration 1: 478.8 GFLOP/s
Iteration 2: 829.9 GFLOP/s
Iteration 3: 830.1 GFLOP/s
```

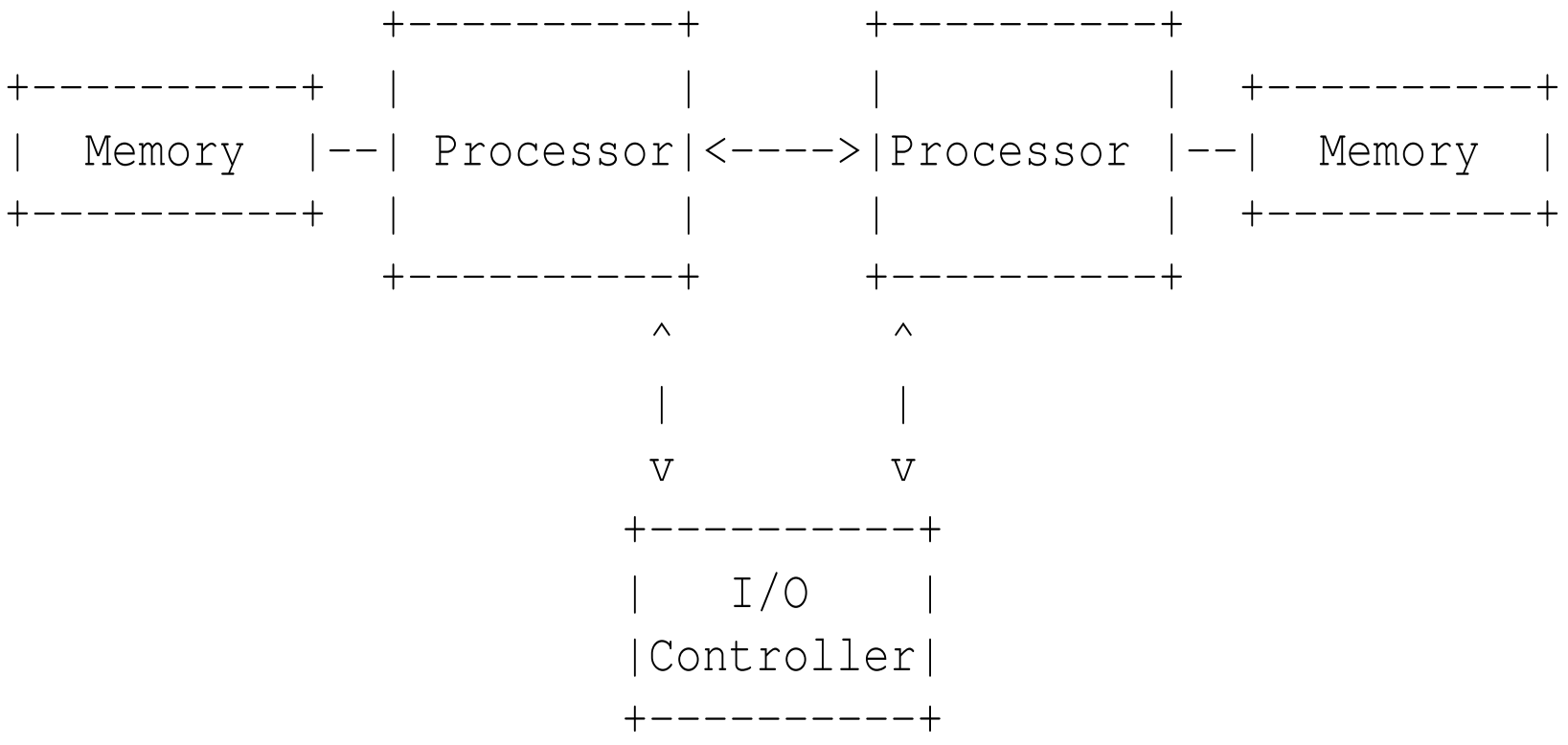
On joker

Note irregular timings when threads are not bound

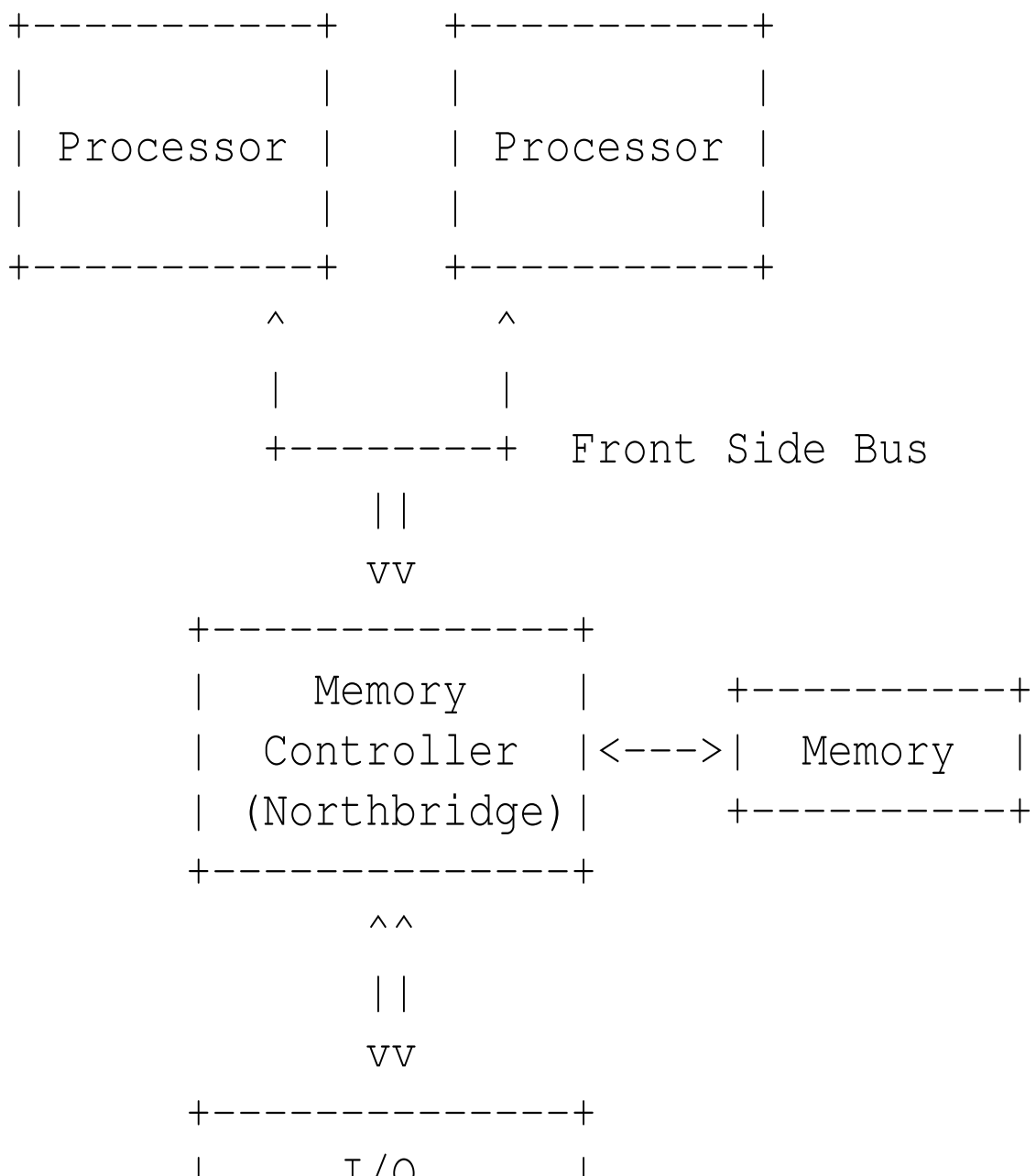
```
joker:~/cse6230/samples2$ ./a.out
Iteration 1: 310.7 GFLOP/s
Iteration 2: 370.2 GFLOP/s
Iteration 3: 400.8 GFLOP/s
Iteration 4: 424.4 GFLOP/s
Iteration 5: 407.6 GFLOP/s
Iteration 6: 435.0 GFLOP/s
Iteration 7: 372.4 GFLOP/s
Iteration 8: 373.4 GFLOP/s
Iteration 9: 369.4 GFLOP/s

joker:~/cse6230/samples2$ KMP_AFFINITY=scatter ./a.out
Iteration 1: 599.2 GFLOP/s
Iteration 2: 757.3 GFLOP/s
Iteration 3: 756.9 GFLOP/s
Iteration 4: 757.1 GFLOP/s
Iteration 5: 757.0 GFLOP/s
Iteration 6: 757.0 GFLOP/s
Iteration 7: 757.1 GFLOP/s
Iteration 8: 757.0 GFLOP/s
Iteration 9: 757.1 GFLOP/s
```

Non-Uniform Memory Access (NUMA)

- Memory access between processor core to main memory is not uniform
- Memory resides in separate regions called *locality domains* or *nodes*
- For highest performance, cores should only access memory in its nearest locality domain



Compare to UMA design (2008 and earlier)



Non-Uniform Memory Access (NUMA)

- NUMA design can give higher aggregate bandwidth to memory
- But memory access is now non-uniform

View the NUMA structure (Haswell)

```
joker:~/cse6230$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 20 21 22 23 24 25 26 27 28 29
node 0 size: 32651 MB
node 0 free: 14883 MB
node 1 cpus: 10 11 12 13 14 15 16 17 18 19 30 31 32 33 34 35 36 37 38 39
node 1 size: 32768 MB
node 1 free: 29266 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

How does the OS decide where to put data?

Page placement by first touch (default)

- A page is placed in the locality region of the processor that first touches it (not when memory is allocated)
- If there is no memory in that locality domain, then another region is used
- Other policies are possible (e.g., set preferred locality domain, or round-robin, using numactl)

```
joker:~/cse6230$ numactl --show
policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ... 39
cpubind: 0 1
nodebind: 0 1
membind: 0 1
```

View the NUMA structure (KNL)

```
available: 8 nodes (0-7)
...
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  21  21  21  31  41  41  41
  1:  21  10  21  21  41  31  41  41
  2:  21  21  10  21  41  41  41  31
  3:  21  21  21  10  41  41  31  41
  4:  31  41  41  41  10  41  41  41
  5:  41  31  41  41  41  10  41  41
  6:  41  41  41  31  41  41  10  41
  7:  41  41  31  41  41  41  41  10
```

Ref: http://colfaxresearch.com/knl-numa/#lst:numactl-h

NUMA effects example: numa1.c

```
a = malloc(N*sizeof(double));
b = malloc(N*sizeof(double));
c = malloc(N*sizeof(double));

for (i=0; i<N; i++)
  a[i] = b[i] = (double) i;

t1 = get_walltime();
#pragma omp parallel num_threads(2)
{
  #pragma omp for schedule(static)
  for (i=0; i<N; i++)
    c[i] = a[i] + b[i];
}
t2 = get_walltime();
```

numa2.c

```
a = malloc(N*sizeof(double));
b = malloc(N*sizeof(double));
c = malloc(N*sizeof(double));

#pragma omp parallel for num_threads(2)
for (i=0; i<N; i++)
  a[i] = b[i] = (double) i;

t1 = get_walltime();
#pragma omp parallel for num_threads(2)
for (i=0; i<N; i++)
  c[i] = a[i] + b[i];
t2 = get_walltime();
```

NUMA effects

```
joker:~/cse6230/samples2$ KMP_AFFINITY=scatter ./numa1
time: 0.373438

joker:~/cse6230/samples2$ KMP_AFFINITY=scatter ./numa2
time: 0.287573
```

NUMA effects

```
joker:~/cse6230/samples2$ numactl -H | grep free
node 0 free: 14705 MB
node 1 free: 29227 MB

# while running numa1
node 0 free: 13941 MB   ( 764 MB used)
node 1 free: 25404 MB   (3823 MB used)


# while running numa2
node 0 free: 12415 MB   (2290 MB used)
node 1 free: 26930 MB   (2297 MB used)
```

Thread affinity for this example

```
joker:~/cse6230/samples2$ KMP_AFFINITY=compact ./numa1
time: 0.656967

joker:~/cse6230/samples2$ KMP_AFFINITY=compact ./numa2
time: 0.551195
```

Why is `compact` worse than `scatter` for this example?

Only allocate for locality domain 0

```
node 0 free: 14699 MB
node 1 free: 29227 MB

joker:~/cse6230/samples2$ KMP_AFFINITY=scatter numactl -m 0
time: 0.382106

node 0 free: 10112 MB   (4587 MB used)
node 1 free: 29227 MB

joker:~/cse6230/samples2$ KMP_AFFINITY=scatter numactl -m 0
time: 0.385136

node 0 free: 10112 MB   (4587 MB used)
node 1 free: 29227 MB
```

Some conclusions

- In our example, using multiple threads to initialize memory helps put data into the “right” locality domain
- Beware of `calloc` to initialize memory