Hands-on Session

Manycore Optimization: Multitasking

Feb. 16, 2018

PRESENTED BY:
Lei Huang

Access KNL/SKX nodes and set up

$ ssh username@stampede2.tacc.utexas.edu

$ idev -m 60 -r advanced_manycore_knl

or

$ idev -m 60 -r advanced_manycore_skx

Prompt similar to “c455-002[**knl**]” or c506-014[**skx**]

$ tar -xvf ~train00/lab_many_core_opt.tgz

$ cd lab_many_core_opt

$ ls -l

```
omp_vector.c   omp_prime.c   sharing.c   false_sharing.c
calc_up.c      calc_up.h     heat.c
```

Differences between KNL and SKX

KNL	SKX
1 socket	2 sockets
68 cores	48 cores (24 cores per socket)
4 HT per core	2 HT per core
1.4GHz	2.0 ~ 3.6 GHz (depends on work load)
-xMIC-AVX512	-xCORE-AVX512

Ex 1: Opt Reports, Cache Reuse, & OpenMP

OpenMP can Interfere with Compiler Optimization

- Compiler often reorders loops for efficient cache reuse
- Remember, 100 times faster to get data from cache than memory
- #pragma omp parallel for could disrupt such optimizations
- Let's look at omp_vector.c
- This looks pretty good right?

```
for ( n = 0 ; n < 1 0 0 0 0 ; n + + ) {
# pragma omp parallel for
    for ( i = 0 ; i <N ; i + + )
        for ( j = 0 ; j <N ; j + + )
            a [ i ]+ =B [ i ] [ j ] * c [ j ] ;
}
```

Ex 1: Opt Reports, Cache Reuse, & OpenMP

1. Compile without OpenMP (you will get a warning, it's ok)

```
$ icc omp_vector.c -o vector -xhost
```

2. Run it and note runtime/flops

```
./vector
```

3. Compile with OpenMP

```
$ icc omp_vector.c -o omp_vector -xhost -qopenmp
```

4. set number of threads to 1 (default)

```
$ export OMP_NUM_THREADS=1
```

5. Run it and note runtime and flops

```
./omp_vector
```

6. Oh noooo! What happened?
7. try increasing thread number and running

Ex 1: Opt Reports, Cache Reuse, & OpenMP

At least throwing a bunch of threads at the problem helps

1. But why is 1 thread of OpenMP much slower than no OpenMP?
2. Compile without OpenMP and qopt-report=5

$ icc omp_vector.c -o vector -xhost -qopt-report=5

3. open omp_vector.optreport and inspect
4. Notice line

remark #25444: Loopnest Interchanged: (1 2 3) --> (2 1 3)

5. Compile with OpenMP and qopt-report

$ icc omp_vector.c -o omp_vector -xhost -qopenmp -qopt-report=5

6. Is that line still there?

Ex 1: Opt Reports, Cache Reuse, & OpenMP

The compiler did not optimize nested loops because of OpenMP directive

1. Oops!
2. Let’s reorder loops by hand

```
# pragma omp parallel for private(i, j, n)
for ( i = 0 ; i <N ; i + + ) {
   for ( n = 0 ; n < 10000 ; n + + )
      for ( j = 0 ; j <N ; j + + )
         a [ i ] +=B [ i ] [ j ] * c [ j ] ;
}
```

3. Compile and rerun. Vary number of threads from 1 to 272

Ex 2: Load Balancing in OpenMP

We'll look at a code with unequal work per iteration

1. omp_prime.c will compute the number of primes between 0 and 10 million
2. As *n grows, more work is required for prime test*

```
# pragma omp parallel for schedule (static) reduction ( + : j )
for ( n = 0 ; n <N ; n + + ) {
        if ( is_prime(n) )
            j + + ;
}
```

3. Compile omp_prime.c with openmp
4. Run with 4,8,68 threads. Record run times
5. Modify to use dynamic scheduling
6. Run with 4,8,68 threads: Record run times
7. Does dynamic scheduling scale better?
8. Vary chunk sizes. Can you improve the performance with chunk size?

```
#pragma omp parallel for schedule(dynamic,chunk_size) reduction(+:j)
```

Ex 3: Sharing Data

sharing.c looks at the effects of Sharing Data and Cache Locality

- Variable a is incremented by each thread in sharing.c
- Must be synchronized with atomic: atomic forces data to be consistent between threads

```
#pragma omp for schedule(static,1)
for (i = 0; i < N; i++) {
#pragma omp atomic
    a++;
}
```

1. compile with icc sharing.c -o sharing -qopenmp -O0 and run.
(-O0 prevents compiler tricks) Record the timing
2. set number of threads to 2 (export OMP_NUM_THREADS=2)
3. use numactl to run on cores 0,68 and 0,1 and 0,2 and record the timing
4. Why are the runtimes different?
5. recall: cores 0,68 share L1, cores 0,1 share L2, cores 0,2 share memory

Ex 4: False Sharing Data

false_sharing.c

- Examine the source code
- 2 different loops for accumulating in parallel
- first loop is naive approach, second loop is smarter
- Each thread accumulates it’s own portion of the iterations
- compile and run

```
$ icc false_sharing.c -o false_sharing -O0 -qopenmp
$ export OMP_NUM_THREADS=1
$ ./false_sharing
```

- Repeat with different thread counts. Which approach scales better?
- What happens to scaling if you set OMP_PROC_BIND=close or spread?

Ex 5: Parallelize Heat Equation

We will parallelize heat.c

- By adding OpenMP and Vectorization
- Solves steady state of 2D heat equation
- Left wall is held constant at 1, other walls are held at 0
- stencil update scheme is defined in calc_up.c

```
void calc_up( int x, int y, int Nx, int Ny, double u [Nx][Ny], double up[Nx][Ny] ) {
    up[x] [y] = u[x][y] + 0.01*(u[x-1][y] + u[x+1][y] - 2*u[x][y] ) +
    0.01*(u[x][y-1]+u [x][y+1]-2*u[x][y] ) ;
}
```

Ex 5: Parallelize Heat Equation

Measure baseline performance w/o parallelization, then add OpenMP

1. Compile calc_up.c

   $ icc calc_up.c -c

2. Compile and run heat.c

   $ icc calc_up.o heat.c -o heat -xhost

3. Run, then record time and sum (sum is to check correctness)
4. Open heat.c and take a look
   - 4.1 I'm only timing t loop (time to solution)
   - 4.2 Where might make sense to add OpenMP parallelization?
5. Add OpenMP pragmas and compile with -qopenmp -qopt-report=5
   - 5.1 hint: 2 places work really well
   - 5.2 you might need to make your iteration variables private (check sum!)
6. Record time at different thread counts and ensure sum is unchanged

Ex 5: Parallelize Heat Equation

Add vectorization through inlining

1. You should see a huge difference with more threads
2. What about vectorization though?
   2.1 take a look at heat.optrpt
   2.2 Was every loop vectorized?
3. Let’s vectorize calc_up with inlining
4. Cross-file inline it:

```
$ icc calc_up.c -ipo -c
$ icc calc_up.o heat.c -o heat -xhost -ipo -qopenmp -qopt-report
```

5. open ipo_out.optrpt and inspect inline and vectorization sections
6. now run heat at different thread count and compare

Ex 5: Parallelize Heat Equation

- We saw some nice speedup: 17s **à** *0.05s !*
- You would see a bigger difference for larger problem