Xeon Phi Processor Tuning (1)

Kevin Olson Ning Li

May 8, 2017

nag®
Experts in numerical algorithms
and HPC services

Introduction

- This marks the point where we start tuning
- Previously focused on parallelism
- Now we focus on throughput
- Also called "time-to-solution", the time it takes to get your answer

Introduction

- Throughput is the processor's capability to do something over time
- Where "something" is usually one of:
  - Bandwidth: Bytes of data from memory per second
  - FLOPS: Floating point operations per second

Bandwidth

- Part 1 focuses on bandwidth, Part 2 will focus on FLOPS
- Huge number of technical applications are bandwidth bound
- Simply because memory operations are expensive

A Bandwidth-bound Program

For example the code below often analyzed as requiring n operations (additions)

```
for(int i=0;i<n;i++){
  z[i]=y[i]+x[i];
}
```

But it also requires $3n$ memory operations, two reads and a write per iterate.

Relative Latencies

- Arithmetic Latency $< 1ns$
- Memory reference $> 100ns$

Bandwidth

- Nothing can really improve the latencies
- But hardware can offer more bytes per memory op
- This is how we overcome the cost

Bandwidth

- Most dual-socket config. servers can offer $100GB/s$
- Xeon Phi processor also can achieve this
- Actually it can achieve much more, as we will see

Performance

Building and running

```
#pragma omp parallel for
for(int i=0;i<n;i++){
  z[i]=y[i]+x[i];
}
```

and computing bandwidth as $3 * 4n/time$ (4 bytes per float,2 reads + 1 write) yields around 80 GB/s.

MCDRAM

MCDRAM

- To improve bandwidth, there is special memory on Xeon Phi processor
- This memory is called MCDRAM
- There is 16GB of it available
- And it can achieve over 400GB/s bandwidth

MCDRAM

- The tricky subject now is: how to use MCDRAM
- There are a few options
- Fortunately none of them have a big setup cost

Operation Modes

- Xeon Phi processor comes with three memory modes
- Can only be set at boot time (unfortunately)
- These simplify managing two memory spaces
  - Normal RAM
  - MCDRAM

Operation Modes: Cache Mode

- In Cache mode all allocations are staged on RAM (don't confuse with 'normal' cache).
- But as requests are made for that data, it is moved into MCDRAM
- This mode works very well for applications which re-use data
  - That does not describe applications of part 1

Operation Modes: Flat

- Flat mode sets up two NUMA nodes
- Usually node 0 is just normal RAM
- and node 1 is MCDRAM

Operation Modes: Flat

- This mainly means that in code, you must specify where to allocate
- libnuma and libmemkind will help us do this

Easiest way to use MCDRAM

if your applications never exceeds 16GB, you don't have to change code at all

```
>> icpc vecaddd.cpp
>> numactl -m1 ./a.out
```

This also uses libnuma, but uses a driver program "numactl" to tell the OS that all allocations will go to node 1, which is MCDRAM in flat mode.

Revisiting Example

Compiling and running

```
typedef NumaAllocator::NumaAllocator<float> na_t;
int mcdram_node=1;
std::vector<float,na_t> z(n,1.0,na_t(mcdram_node));
std::vector<float,na_t> y(n,1.0,na_t(mcdram_node));
std::vector<float,na_t> x(n,1.0,na_t(mcdram_node));
#pragma omp parallel for
for(int i=0;i<n;i++){
  z[i]=y[i]+x[i];
}
```

on KNL gives the same results but a great $450GB/s$ bandwidth. That's over a $4X$ speedup from before.

Revisiting Example

It is also possible to use libnuma directly, as below

```
int mcdram_node=1;
float* z = (float*) numa_alloc_onnode(sizeof(float)*n
float* y = (float*) numa_alloc_onnode(sizeof(float)*n
float* x = (float*) numa_alloc_onnode(sizeof(float)*n
#pragma omp parallel for
for(int i=0;i<n;i++){
  z[i]=y[i]+x[i];
}
numa_free(z,sizeof(float)*n);
numa_free(y,sizeof(float)*n);
numa_free(x,sizeof(float)*n);
```

And for Fortran...

To force allocatable arrays in Fortran to be allocated on MCDRAM:

```
!DIR$ ATTRIBUTES FASTMEM :: x, y, z
double precision, allocatable, dimension(:) :: &
      x, y, z
allocate(x(n))
allocate(y(n))
allocate(z(n))
!$OMP PARALLEL DO
DO i = 1, n
   z(i)=y(i)+x(i)
END DO
!$OMP END PARALLEL DO
deallocate(x)
deallocate(y)
deallocate(z)
```

Another possibility

Only array x will be allocated to MCDRAM.

```
double precision, allocatable, dimension(:) :: &
      x, y, z
!DIR$ FASTMEM
allocate(x(n))
allocate(y(n))
allocate(z(n))
!$OMP PARALLEL DO
DO i = 1, n
   z(i)=y(i)+x(i)
END DO
!$OMP END PARALLEL DO
deallocate(x)
deallocate(y)
deallocate(z)
```

Remarks

- Best Performance: Flat + NUMA allocations
- Medium Performance: Cache mode
- Lowest Performance: RAM only

Results

	GB/s
MCDRAM	1.2s
Cache	0.3s
RAM	0.4s

BLOCKING

Advanced Tuning

- Many applications do more operations per byte
- Not only read in data, do one thing, then throw it away
- In this case it is possible to overcome 16GB limitation

Advanced Tuning

```
for(int64_t i=0;i<n;i++){
  for(int64_t j=0;j<n;j++){
    z[i]+=x[i]*y[j];
  }
}
```

Suppose for example that n above very large.

Advanced Tuning

```
for(int64_t i=0;i<n;i++){
  for(int64_t j=0;j<n;j++){
    z[i]+=x[i]*y[j];
  }
}
```

There are only n elements per array but there are n^2 operations.

Advanced Tuning

```
int64_t BI=1e9*5/sizeof(float)
for(int64_t i=0;i<n;i+=BI){
  for(int64_t j=0;j<n;j++){
    for(int64_t ii=i;ii<i+BI;ii++)
      z[ii]+=x[ii]*y[j];
  }
}
```

In the above we "block" top loop with BI making sure it is less than 16GB of data.

Advanced Tuning

```
int64_t BI=1e9*5/sizeof(float)
for(int64_t i=0;i<n;i+=BI){
  for(int64_t j=0;j<n;j++){
    for(int64_t ii=i;ii<i+BI;ii++)
      z[ii]+=x[ii]*y[j];
  }
}
```

Therefore in new inner loop at each j iterate we re-use significant amount of data.

Advanced Tuning

- This "blocking" can work in cache mode or flat mode
- but in flat mode you must manage the memory yourself
- That is left as exercise

Advanced Tuning

- Blocking is extremely useful
- We will see it in more depth in tuning: part 2
- But it is here just to show that 16 GB is not a hard limit

HYPERTHREADING

Advanced Tuning

- Since memory ops are so expensive, special hardware helps too
- One way is through hyperthreading

Hyperthreading

- In addition to the 72 cores, each core has 4 "hardware threads"
- These threads can be seen as concurrent by programmer
- But they share resources, and so this does not mean $4X$ speedup

Hyperthreading

- Hyperthreading is complex, but generally benefits when:
  - There is reasonable amount of compute
  - But: program still stalling for data as well

Hyperthreading

```
int ncores=72;
#pragma omp parallel for nthreads(1*ncores) \
  proc_bind(spread)
for(int64_t i=0;i<n;i+=BI){
  for(int64_t j=0;j<n;j++){
    for(int64_t ii=i;ii<i+BI;ii++)
      z[ii]+=x[ii]*y[j];
  }
}
```

Previous example without hyperthreading

Hyperthreading

```
#pragma omp parallel for nthreads(2*ncores) \
  proc_bind(spread)
for(int64_t i=0;i<n;i+=BI){
  for(int64_t j=0;j<n;j++){
    for(int64_t ii=i;ii<i+BI;ii++)
      z[ii]+=x[ii]*y[j];
  }
}
```

And now two hardware threads per core

Hyperthreading: Results

	Relative Time
1X/core	1.0
2X/core	1.0
4X/core	1.0

Thread Affinity

- Hyperthreading does create a new issue
- By using more threads than cores, must decide where threads go
- OS decides usually, but you can use thread affinity to force it

Thread Affinity: Example

```
#pragma omp parallel for nthreads(2*ncores) \
  proc_bind(spread)
for(int64_t i=0;i<n;i+=BI){
  for(int64_t j=0;j<n;j++){
    for(int64_t ii=i;ii<i+BI;ii++)
      z[ii]+=x[ii]*y[j];
  }
}
```

Note proc_bind(spread) here. It ensures threads utilize unused cores before unused hardware threads.

Thread Affinity

- Always check affinity before making conclusions
- Bad affinity setting result in bad performance
- Sometimes the defaults are wrong also!

Conclusions

- Here you saw some tuning strategies
- Working with MCDRAM
  - Using it for bandwidth bound problems
  - Working within the 16GB limits
  - Understanding flat versus cache
- Hyperthreading and affinity

Practical Exercise

USING MCDRAM and BLOCKING A CODE