OpenMP

How to NOT screw up parallelization

In this session

- What‘s OpenMP?
- Brief overview of the features.
- How it works under the hood?
- Naive approach doesn‘t always work!
- Affinity and cache utilization

Hello world!

```
#include<stdio.h>
void main() {
	#pragma omp parallel  {
		printf("Hello world\n");
	}
}
```

OpemMP History

- OpenMP 2 – highly regular loops
- OpenMP 3 – tasks
- OpenMP 4 – SIMD, accelerators, affinity, atomics

#pragma omp

```
#pragma omp master
printf("I‘m a lonely master\n");
```

Runtime functions

- omp_get_thread_num()
- omp_get_num_threads()
- omp_get_wtime()
- omp_in_parallel()

Environment variables

- OMP_NUM_THREADS
- OMP_PLACES
- OMP_PROC_BIND
- GOMP_CPU_AFFINITY
- KMP_AFFINITY

Where OMP works

Good

- Any shared memory
- GPU

Bad

- Cluster

There were multiple OpenMP cluster implementations from Intel

How it works?

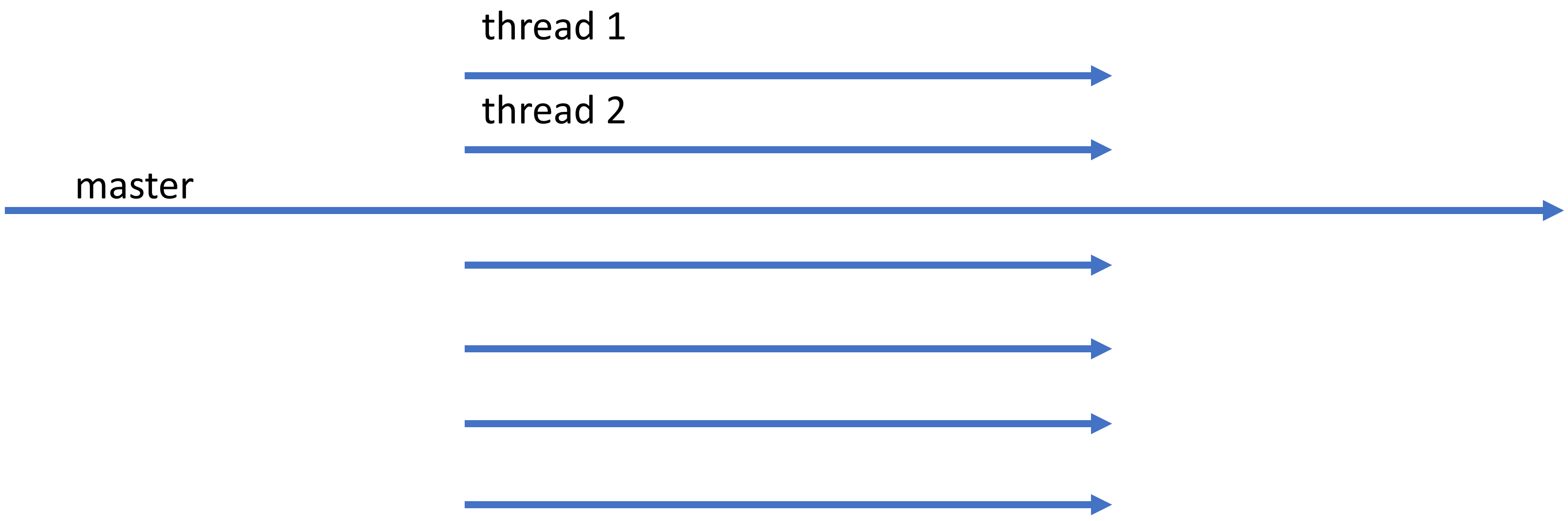
How do we look into it?

- export OMP_DISPLAY_ENV=true/verbose
- KMP_AFFINITY=verbose for Intel OMP
- gcc -fdump-tree-all … to see Gimple code
- gcc –S
- Source code of libgomp

Inside Hello World

```
//Generated code
main._omp_fn.0 (void * .omp_data_i) {
printf ("Hello world!\n");
}
main () {
//3 – number of threads
__builtin_GOMP_parallel (main._omp_fn.0, 0B, 3, 0);
}
```

Fork-join



OMP_WAIT_POLICY controls the behavior of the threads

OMP_WAIT_POLICY

... is a potential issue for FreeLibrary or dlclose

Parallel sections

```
#include <stdio.h>
void main() {
  #pragma omp parallel sections
  {
          #pragma omp section
          {printf("Section1\n");}
          #pragma omp section
          {printf("Section2\n");}
  }
}
```

Inside section

- Fork multiple threads
- Each thread picks on a yet unfinished section.

Parallel for loop

```
void fun(double *data, int n) {
    #pragma omp parallel for
    for (int i = 0; i < n; i++)
        data[i] += data[i]*data[i];
}
```

Inside loop code

Look in the intermediate

Basically a pretty CUDA-like code

- Ask your thread‘s id, ask the total number of threads
- Calculate your share of the array.
- Handle it

Atomic instruction

```
#pragma omp atomic
a += 2;

//No cmp and exchange, though
```

Inside atomic

Replaces the call with e.g. __atomic_fetch_add_4

Data model

	Passed In	Passed Out
Private		
First private	Y	
Last private		Y
Shared	Y	Y

Critical section

```
#pragma omp critical
printf("I'm critical\n");
```

Inside critical section

```
__builtin_GOMP_critical_start ();
  printf ("I\'m critical");
  __builtin_GOMP_critical_end ();
```

The critical section is shared between the threads

Barrier

```
//Wait till all spawned threads are done
#pragma omp barrier
```

Single only

```
#pragma omp single
printf("I‘m a single!\n");
```

Inside single

- Check‘s if somebody did it - __builtin_GOMP_single_start()
- If not – run it and set the flag

Master only

```
#pragma omp master
printf("I‘m a master!\n");
```

Inside master

- Just check if thread id is 0

Single vs master

Single

- Has a barrier
- Executes on any thread

Master

- No barrier
- Executes on 0-thread aka master

SIMD

```
#pragma omp simd
  for(int i=0;i<N;++i) {
    a[i] += b[i];
  }
```

Inside SIMD

movss(%rcx,%rax,4), %xmm0

addss %xmm1, %xmm0

//To be honest, the compiler could do it on it‘s own

Tasks

```
#pragma omp parallel
{
#pragma omp single
    {
    #pragma omp task
    printf("hello world\n");

    #pragma omp taskwait

    #pragma omp task
    printf("hello again!\n");
    }
}
```

False sharing

```
int sum_a(void) {
    int s = 0;
    for (unsigned i = 0; i < N; ++i)
        s += f.x;
    return s;
}
void inc_b(void) {
    for (unsigned i = 0; i < N; ++i)
        ++f.y;
}
//3x faster, if y aligned out of cache page
```

False sharing

Multiple processors update the same cache line – MESI

Intel compiler claims to be especially smart finding such issues.

Open MP 5 – Data Affinity

```
void task_affinity() {
double* B;
#pragma omp task shared(B) affinity(A[0:N])
{
	B = init_B_and_important_computation(A);
}
#pragma omp task firstprivate(B) affinity(B[0:N])
{
	important_computation_too(B);
}
#pragma omp taskwait
}
```

What‘s actually c5.18xlarge?

lscpu

CPU(s): 72
On-line CPU(s) list: 0-71
Thread(s) per core: 2
Core(s) per socket: 18
Socket(s): 2
NUMA node(s): 2

Mxnet and OpenMP

- Multiple implementations
- Conflict on true one on github
- OMP_DISPLAY_ENV can actually show multiple implementations initialized for **default** build (gcc with mkl_intel)
- Due to that – OMP_PLACES/OMP_PROC_BIND kills performance like a boss

Benchmark setup

- GCC5 + GOMP
- GCC5 + IOMP – aka default build
- C5.18xlarge to run on
- Cmake build

Relevant OpenMP runtimes

- GOMP
- LLVM OMP – can fake GOMP. Actually derived from IOMP
- Intel OMP (close to LLVM OMP) – can fake GOMP
- Whatever MS does.

Multiple OpenMP runtimes at once?

- „Intel MKL-DNN can use Intel, GNU or CLANG OpenMP runtime. Because different OpenMP runtimes may not be binary compatible, it's important to ensure that only one OpenMP runtime is used throughout the application. **Having more than one OpenMP runtime initialized may lead to undefined behavior including incorrect results or crashes**“

Static linking?

„On glibc-based systems, OpenMP enabled applications cannot be statically linked due to limitations of the underlying pthreads-implementation. It might be possible to get a working solution if -Wl,--whole-archive -lpthread -Wl,--no-whole-archive is added to the command line. However, this is not supported by gcc and thus not recommended.“

What‘s actually c5.18xlarge?

numactl –hardware

available: 2 nodes (0-1)
node 0 cpus: 0-17 36-53
node 0 size: 70349 MB
node 0 free: 64388 MB
node 1 cpus: 18-35 54-71
node 1 size: 70427 MB
node 1 free: 40161 MB
node distances:
node   0   1
  0: 10 21
  1: 21 10

NUMA

- Modern systems do first-touch
- Initialize data in parallel. Otherwise one node can access other nodes memory.
- Or use numactl to allocate on different nodes
- OpenMP has no control over it

What‘s actually c5.18xlarge?

cpuinfo

- L1 32 KB (0,36)(1,37)(2,38) …
- L2 1   MB (0,36)(1,37)(2,38)…
- L3 24 MB (0-17,36-53)(18-35,54-71)

OMP_PLACES

- sockets
- threads
- cores
- Specific list

OMP_PROC_BIND

- true
- false
- close
- spread
- master

OMP_PLACES

Close

- Fast synchronization
- Less bandwidht/cache

Spread

- Slower synchronization
- More bandwidth

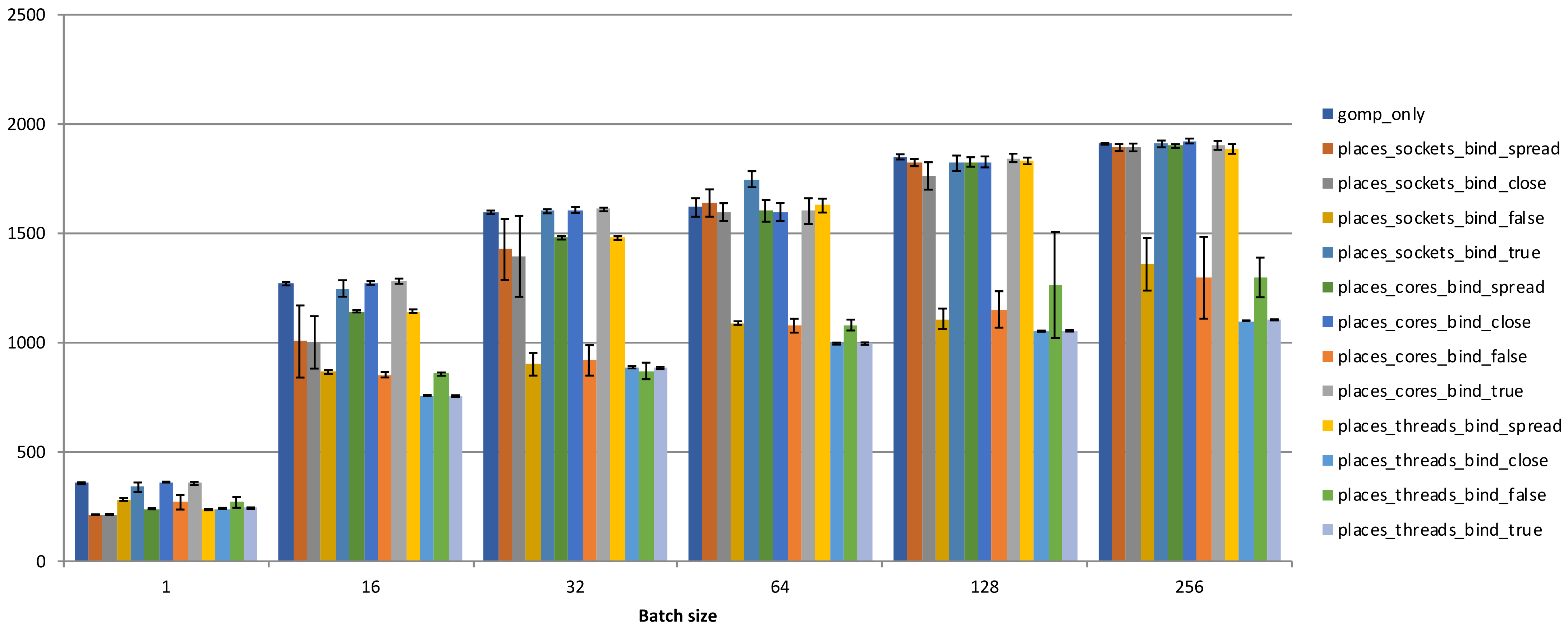
GOMP_CPU_AFFINITY

- Something like 0-71 or 0 3 1-2 4-15:2

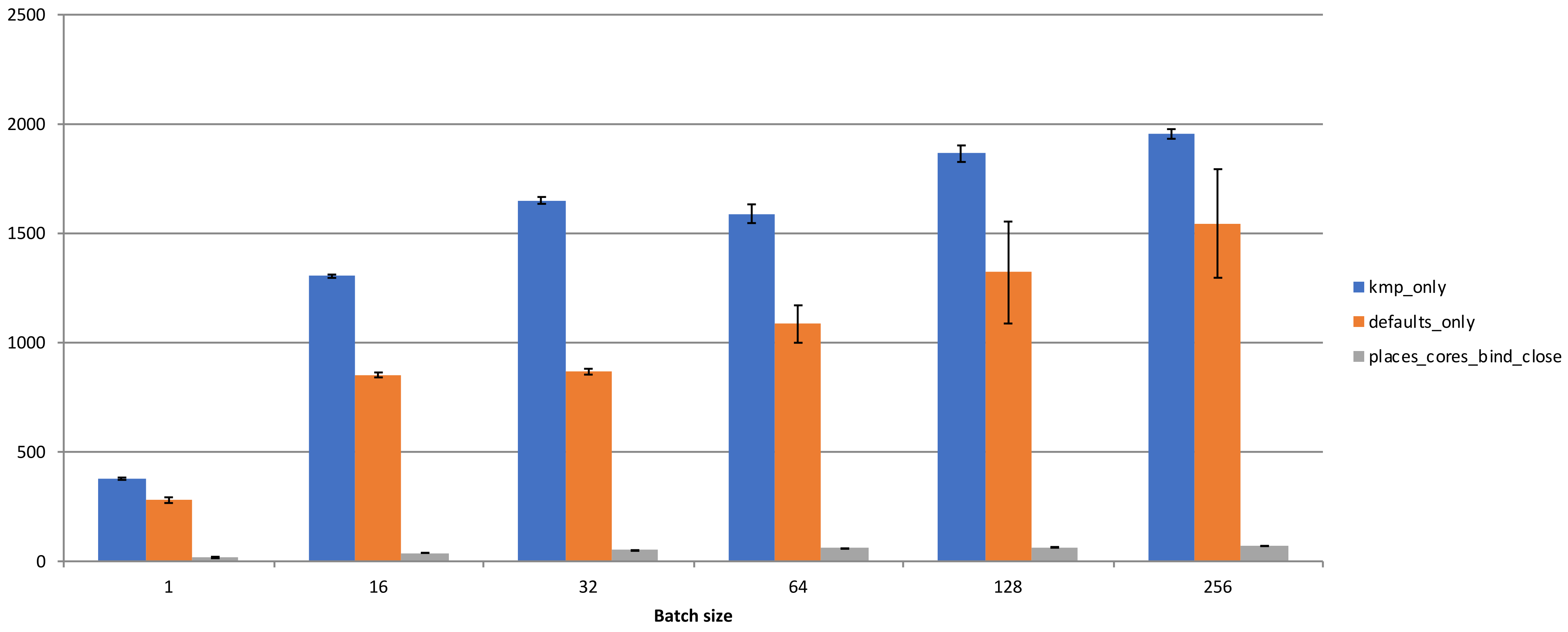
KMP_AFFINITY

- Deprecated by Intel. Use OMP_ until really necessary
- KMP_AFFINITY=verbose is the only viable use case
- **granularity=fine,compact,1,0** seems to reach the optimal performance. compact ~ close

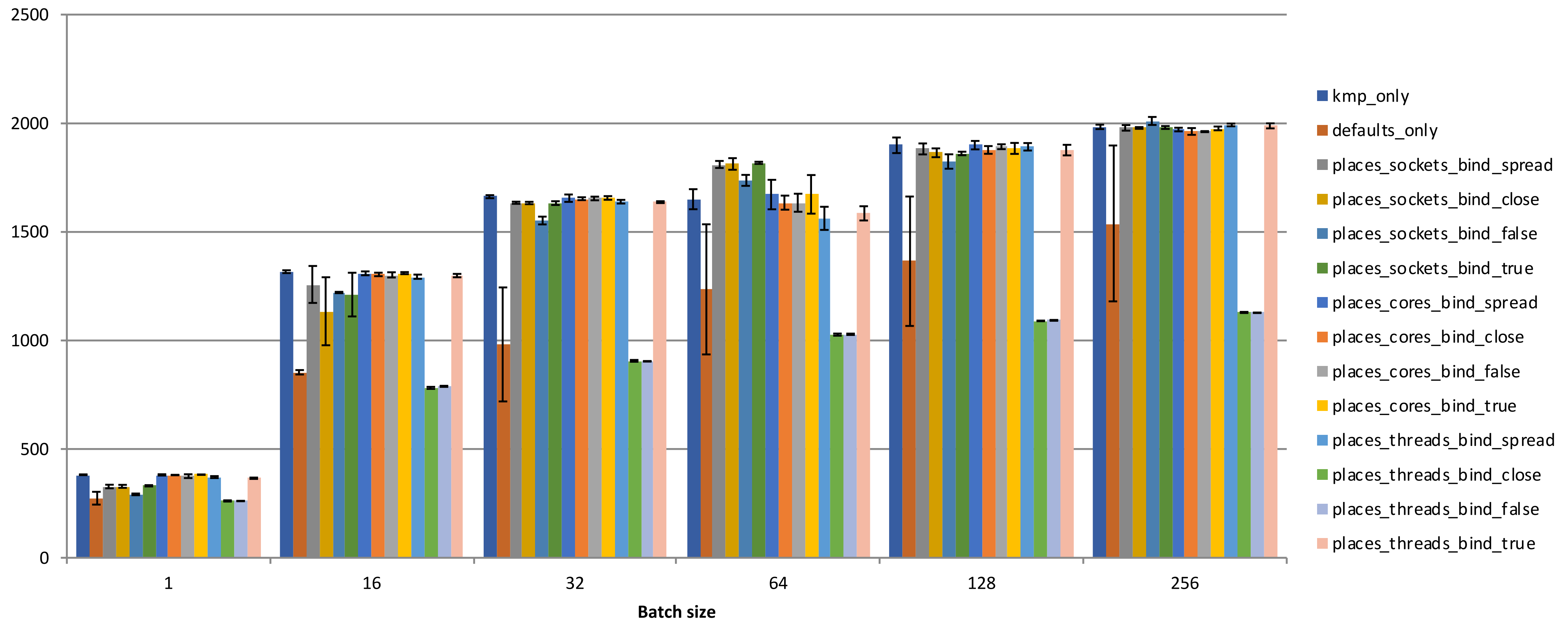
So what‘s the results for GCC?



What about default build?



Fixed default build



What have we learned?

- “Threads“ are not independent processors for HT
- No reason to disable proc-binding
- Mxnet benefits from cores places more than from sockets
- Long-run almost no difference between „good“ configurations

Learned elsewhere

- LLVM OMP is faster than GOMP