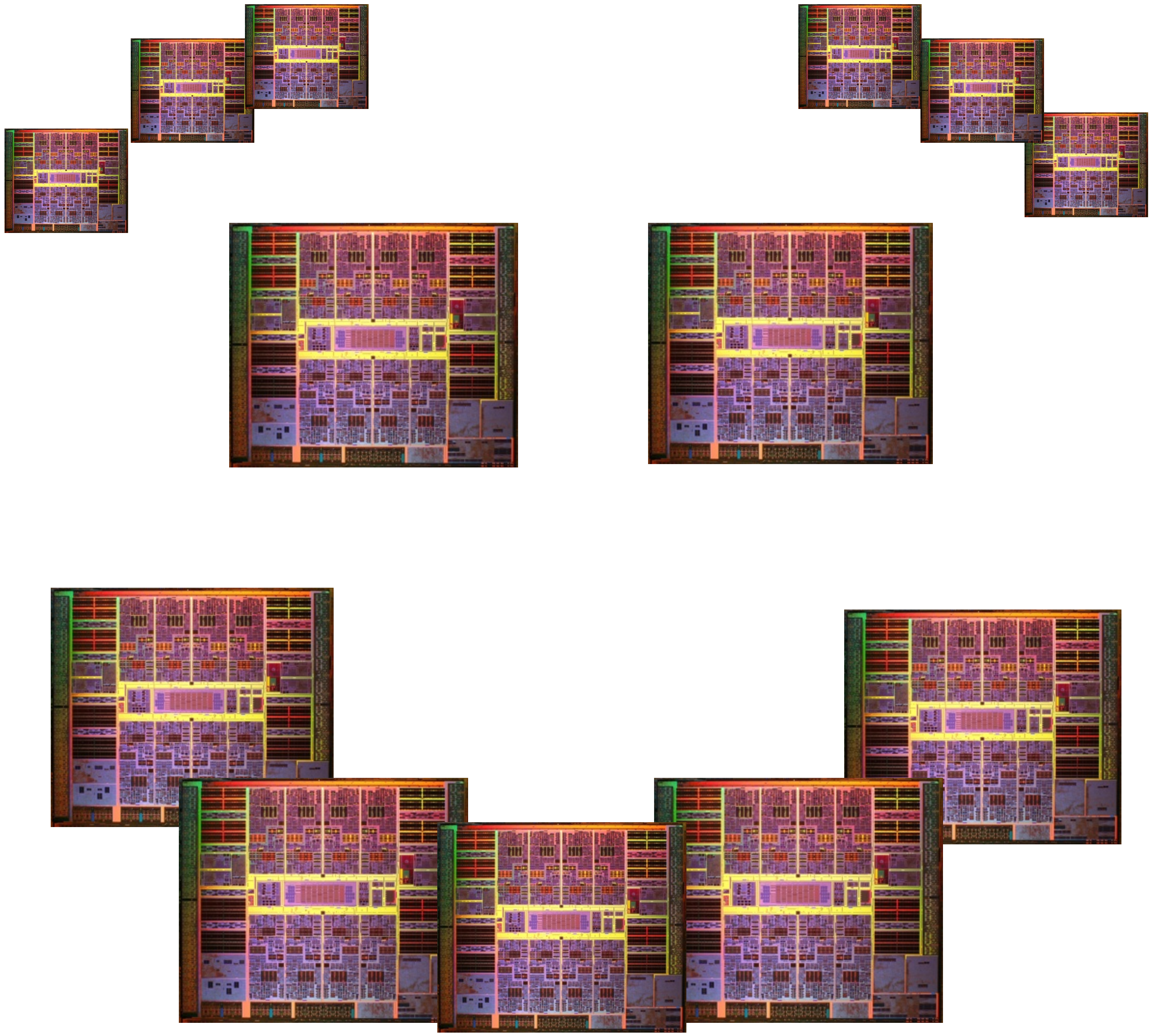
GHC on the OpenSPARC T2

Ben Lippmeier
Australian National University
Haskell Implementors Workshop
2009/08/05

The Project

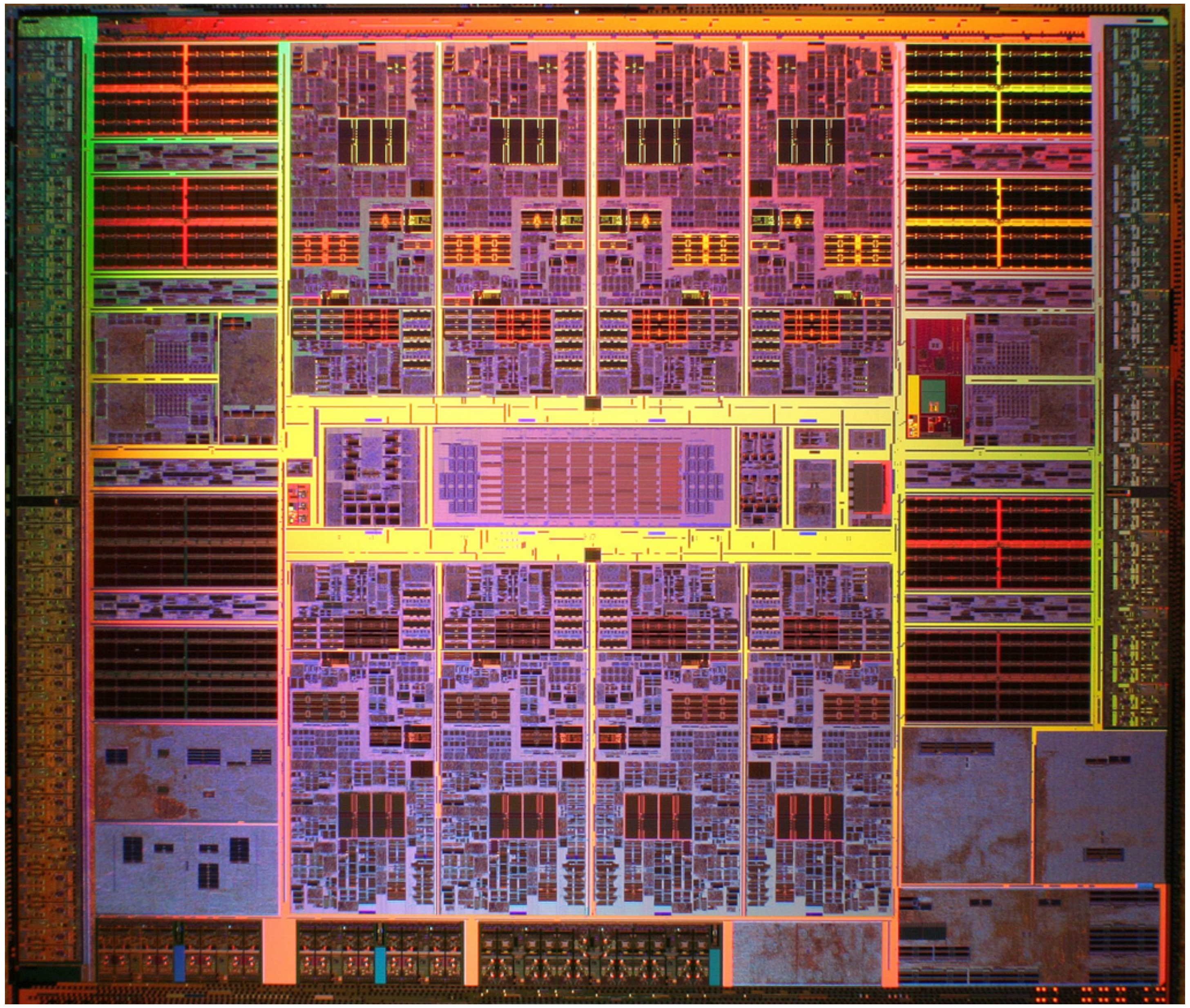
- Funded by Sun Microsystems.
- Organised by:
  - Duncan Coutts, Roman Leshchinskiy, Darryl Gove.
- Make GHC work on SPARC (again)
- Why do we care?



Multicore !!!

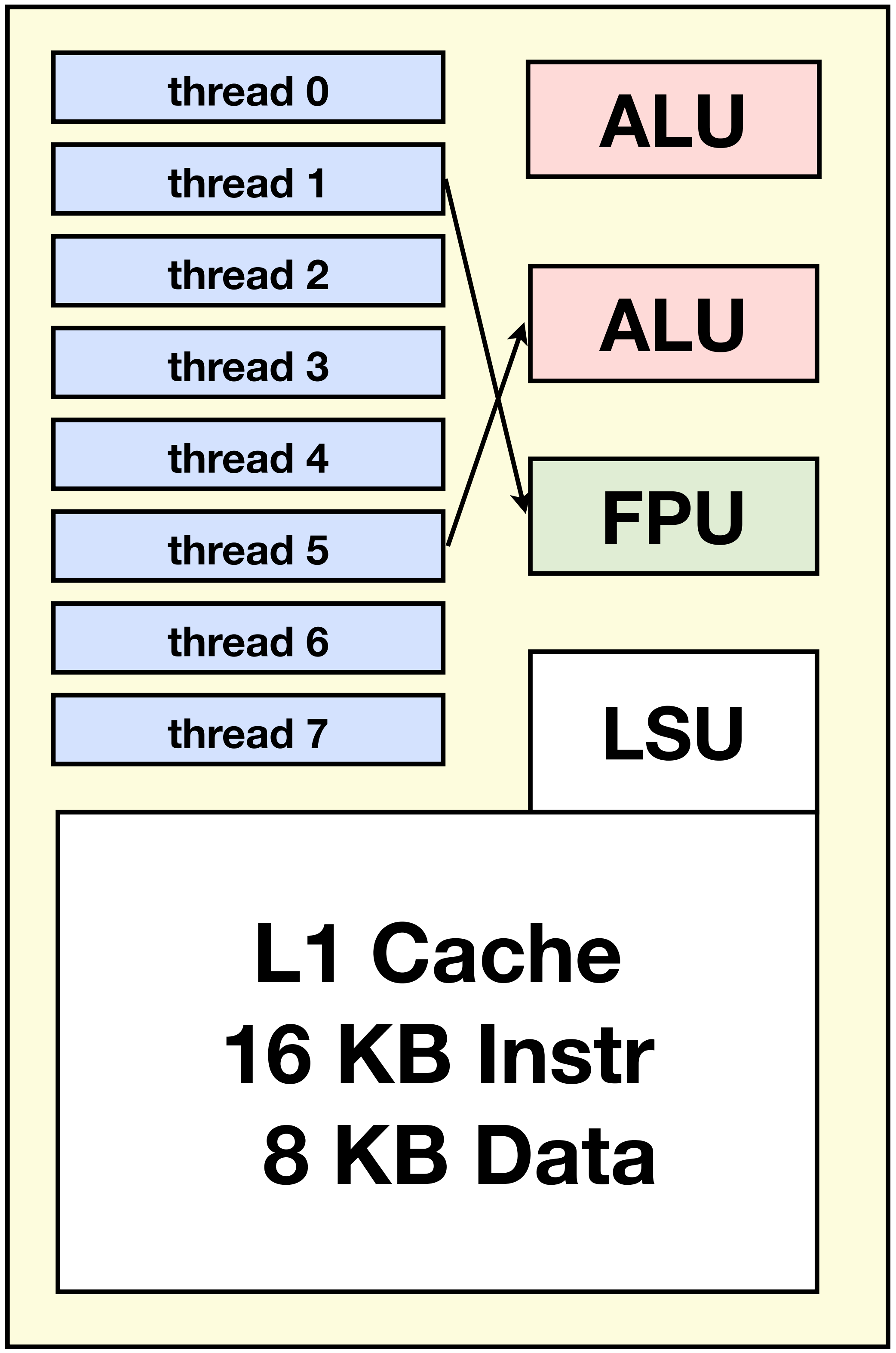
(shared memory symmetric multi-processing)

The OpenSPARC T2



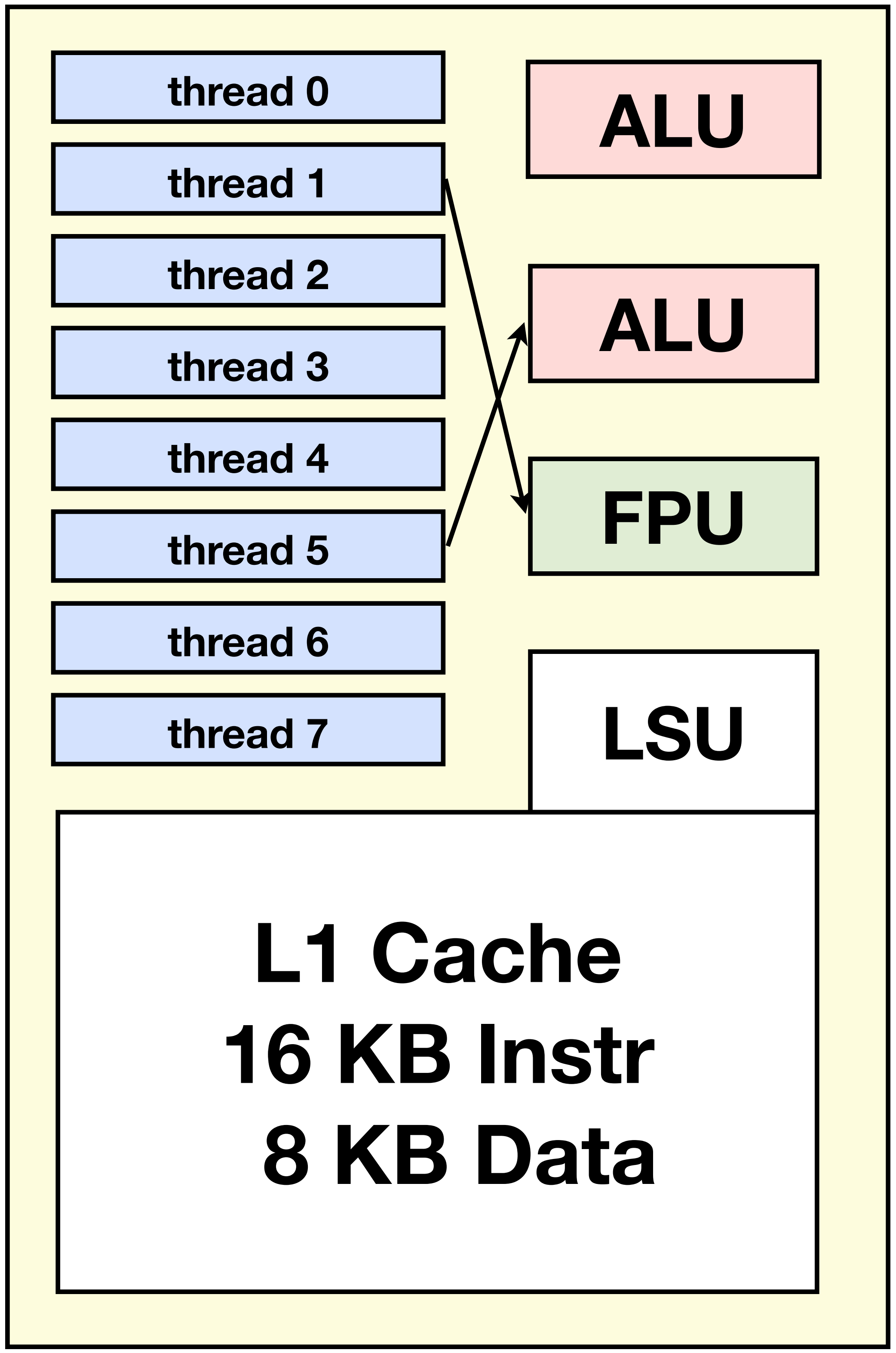
- Released Oct 2007
- 8 cores / processor
- 8 threads / core
- 64 threads / processor
- 4 MB L2 Cache
  16 way associative.
- 1165 MHz

One T2 Core



- Hardware per core:
  - 2 x ALU (Integer + Address)
  - 1 x FPU (Floating Point)
  - 1 x LSU (Load Store Unit)
- 8 stage integer pipeline
- 12 stage floating point pipeline
- No out-of-order execution
- No exploitation of instruction level parallelism (ILP)

One T2 Core



- Each thread has its own register set.
- Two instructions can be dispatched per cycle, each from different threads.
- Threads are intended to stall frequently.
- All threads on a core share the same L1 Cache.

Peak instruction issue rates

OpenSPARC T2

1165 MHz * 2 instrs/core * 8 cores

= 18.64 Gig instrs / s

(in order)

Intel Core2 Duo

1600 MHz * 4 instrs/core * 2 cores

= 12.80 Gig instrs / s

(out of order)

Out-of-order execution doesn't help us much...

```
ld        [%i0+4],   %g1
st        %g1,       [%i3-12]
st        %l2,       [%i3-8]
ld        [%i0+12],  %g1
st        %g1,       [%i3-4]
st        %l1,       [%i3]
add       %i3, -24,  %g1
st        %g1,       [%i0+12]
ld        [%i0+8],   %l1
sethi     %hi(s1rX_info), %g1
or        %g1,       %lo(s1rX_info), %g1
st        %g1,       [%i0+8]
add       %i0, 8,    %i0
and       %l1, 3,    %g1
cmp       %g1,       0
bne       .Lc1Un
```

- Lots of memory traffic
  => Lots of cache miss
- Not much ILP
  (Instr Level Parallelism)

Fixing the Native Code Generator

- GHC has had native code generation for
  - x86
  - x86_64
  - Power PC
  - SPARC
  - Alpha
- All mashed into one module “MachCodeGen.hs”
- Support for various architectures has grown organically.
- Target architecture selected by a series of #ifdefs

#ifdef is not my friend

```
#if i386_TARGET_ARCH || x86_64_TARGET_ARCH

getRegister (CmmMachOp mop [x])
  = case mop of
#if i386_TARGET_ARCH
      MO_F_Neg W32 -> trivialUFCode FF32 (GNEG FF32) x
      MO_F_Neg W64 -> trivialUFCode FF64 (GNEG FF64) x
#endif

...
#if sparc_TARGET_ARCH
getRegister (CmmLit (CmmFloat f W32))
 = do ...

...
#if powerpc_TARGET_ARCH
getRegister (CmmLoad mem pk)
  | not (isWord64 pk)
  = ....
```

#ifdef is not my friend

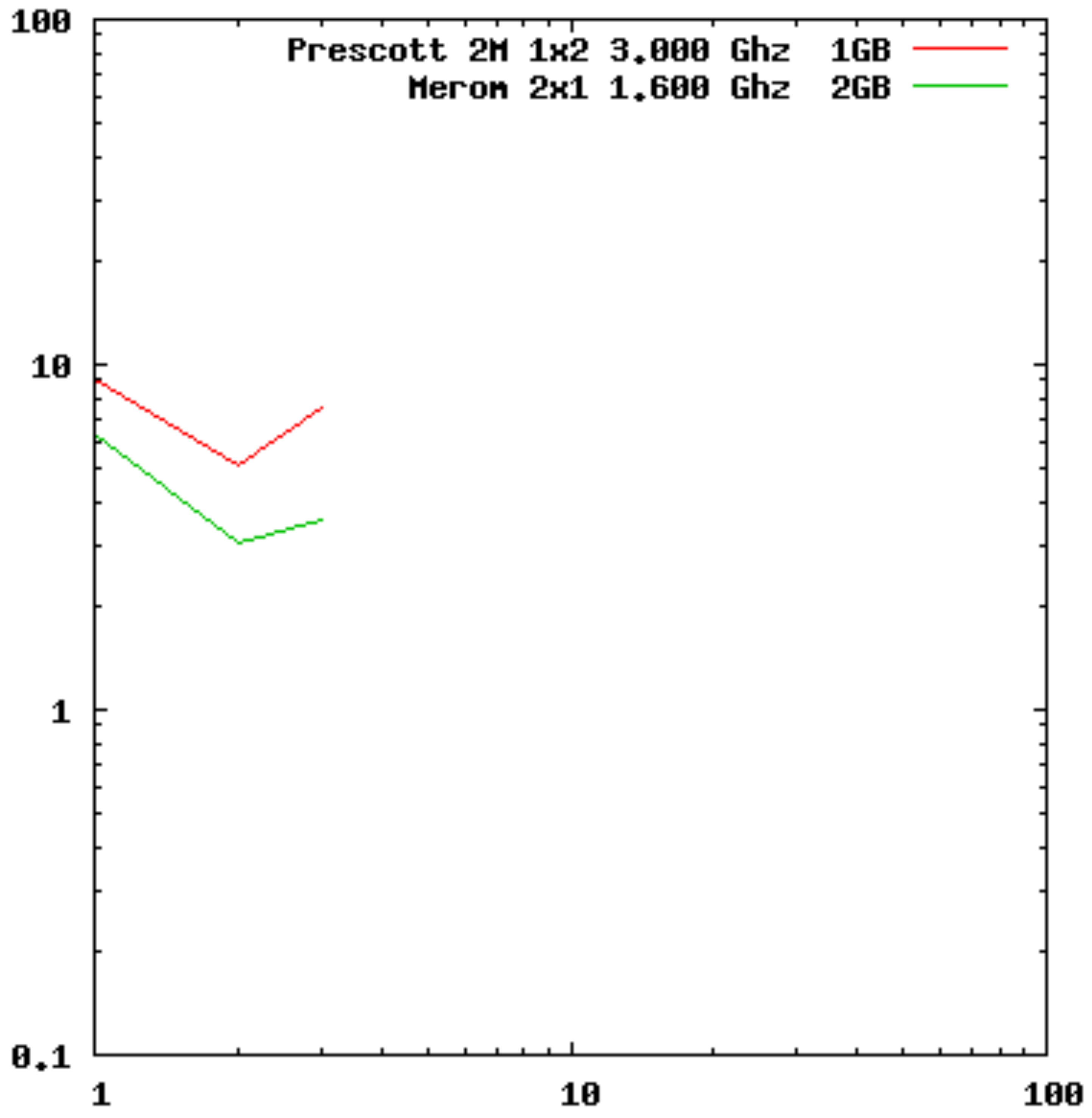
- Hard to work on code for one platform without breaking others.
- All code for all platforms should be compiled all the time.
- Code for SPARC and PPC is now split into its own modules.
- Still need to untangle x86 from x86_64.
- Move towards being a cross-compiler, and eliminate dependency on GCC for bootstrapping.

The Instruction class

```
class   Instruction instr where
 regUsageOfInstr      :: instr -> RegUsage
 patchRegsOfInstr     :: instr -> (Reg -> Reg) -> instr

 isJumpishInstr       :: instr -> Bool
 jumpDestsOfInstr     :: instr -> [BlockId]
 patchJumpInstr       :: instr -> (BlockId -> BlockId)-> instr
 mkJumpInstr          :: BlockId -> [instr]

 mkSpillInstr         :: Reg    -> Int -> Int -> instr
 mkLoadInstr          :: Reg    -> Int -> Int -> instr

 takeDeltaInstr       :: instr -> Maybe Int
 isMetaInstr          :: instr -> Bool

 mkRegRegMoveInstr    :: Reg    -> Reg -> instr
 takeRegRegMoveInstr :: instr -> Maybe (Reg, Reg)
```

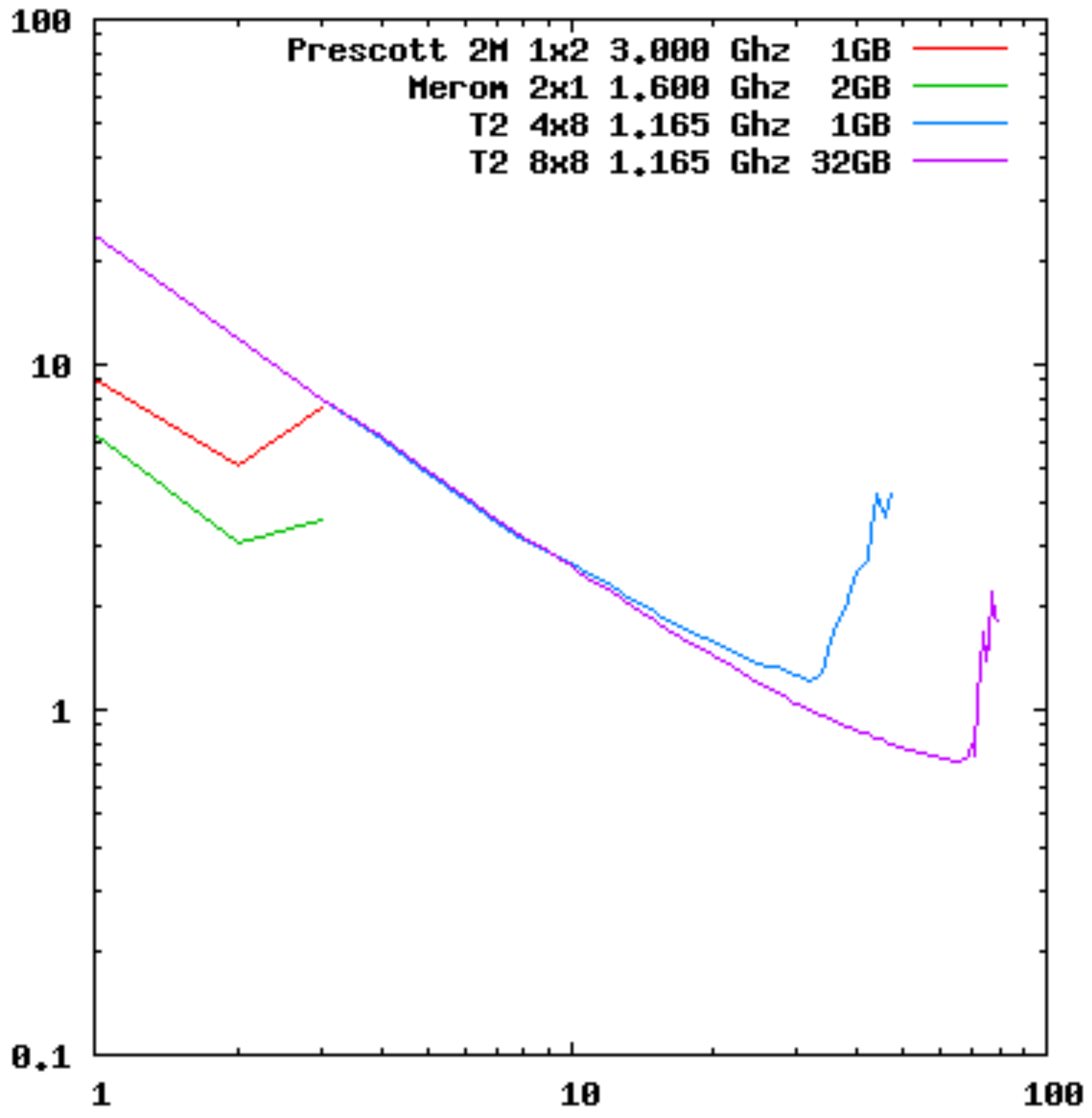
Benchmarking

sumeuler: runtime(s) *vs* number of threads



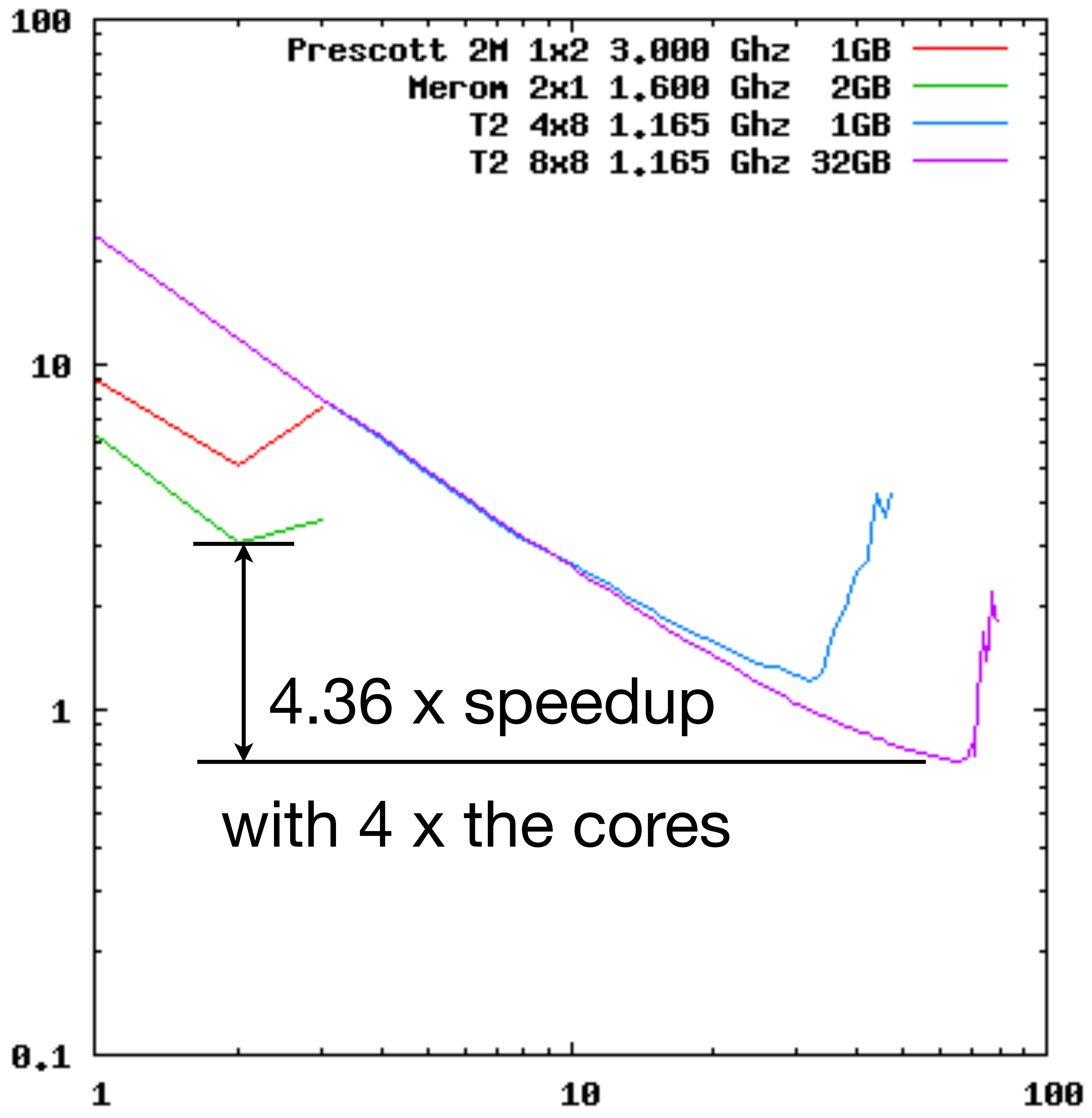
- Embarrassingly parallel benchmark.
- Use Intel processors as the baseline.

sumeuler: runtime(s) *vs* number of threads



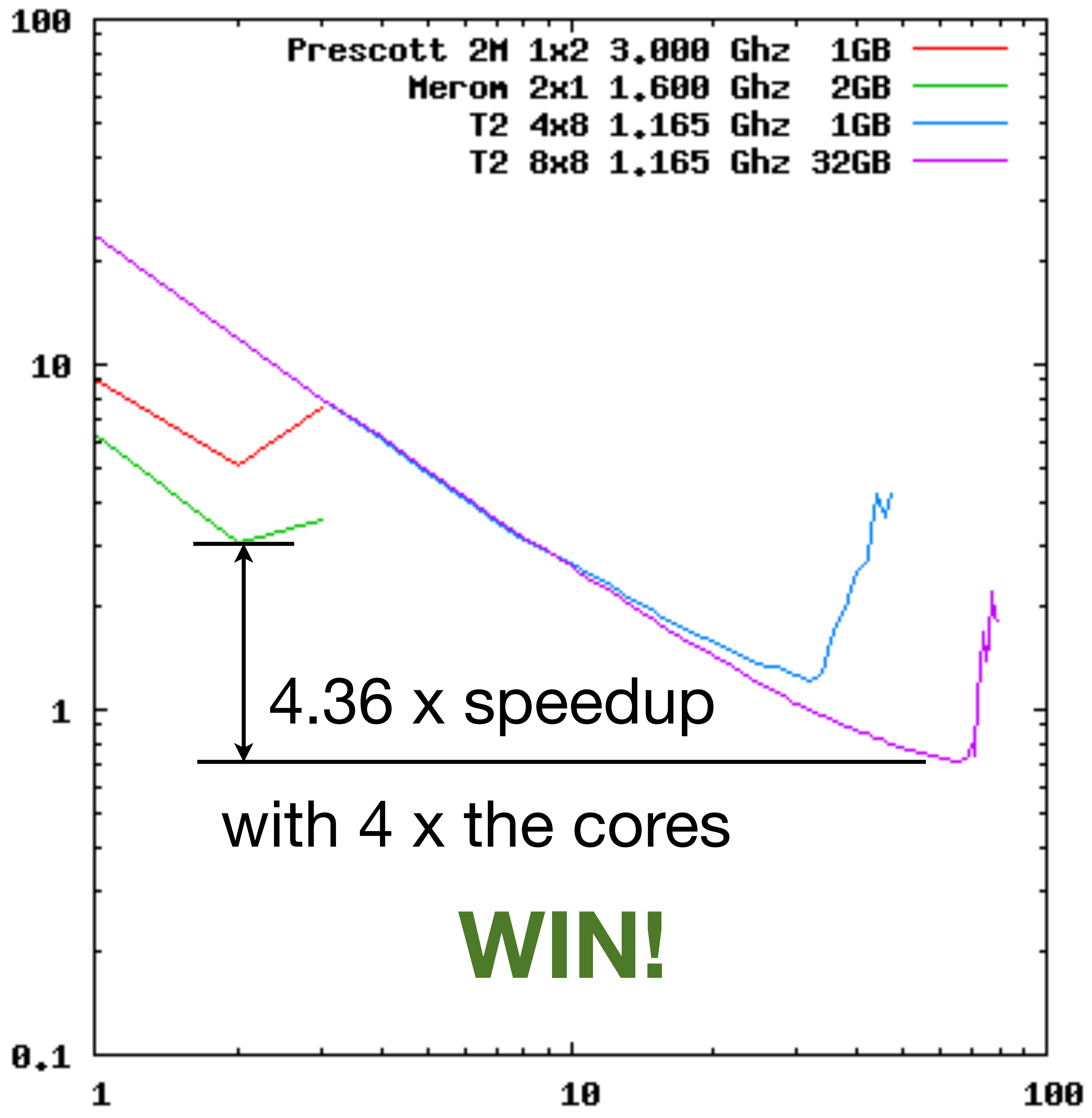
- Embarrassingly parallel benchmark.
- Use Intel processors as the baseline.
- Almost linear speedup until we run out of hardware threads.
- No point using more Haskell threads than hardware threads.

sumeuler: runtime(s) *vs* number of threads



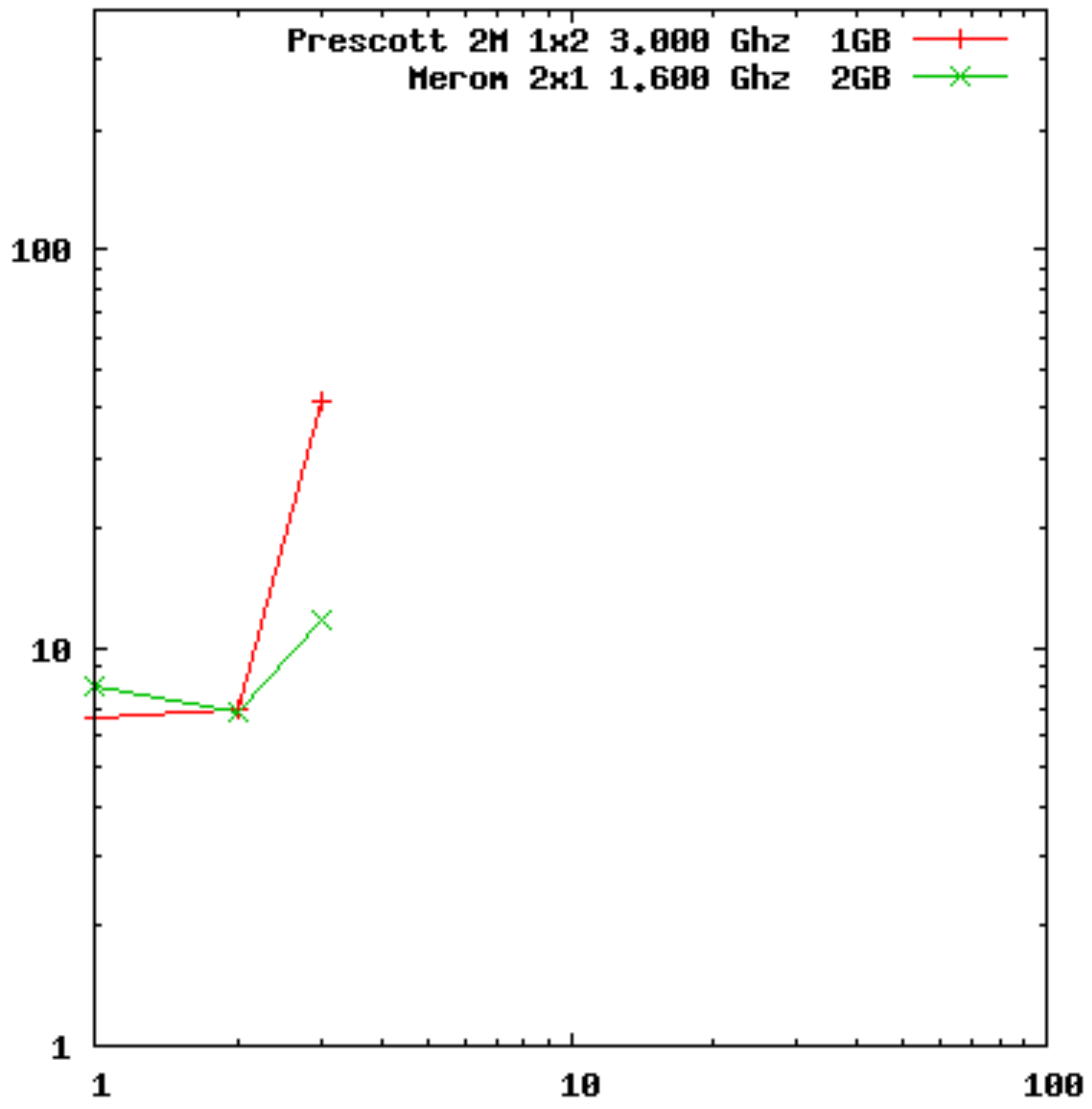
- Embarrassingly parallel benchmark.
- Use Intel processors as the baseline.
- Almost linear speedup until we run out of hardware threads.
- No point using more Haskell threads than hardware threads.

sumeuler: runtime(s) *vs* number of threads



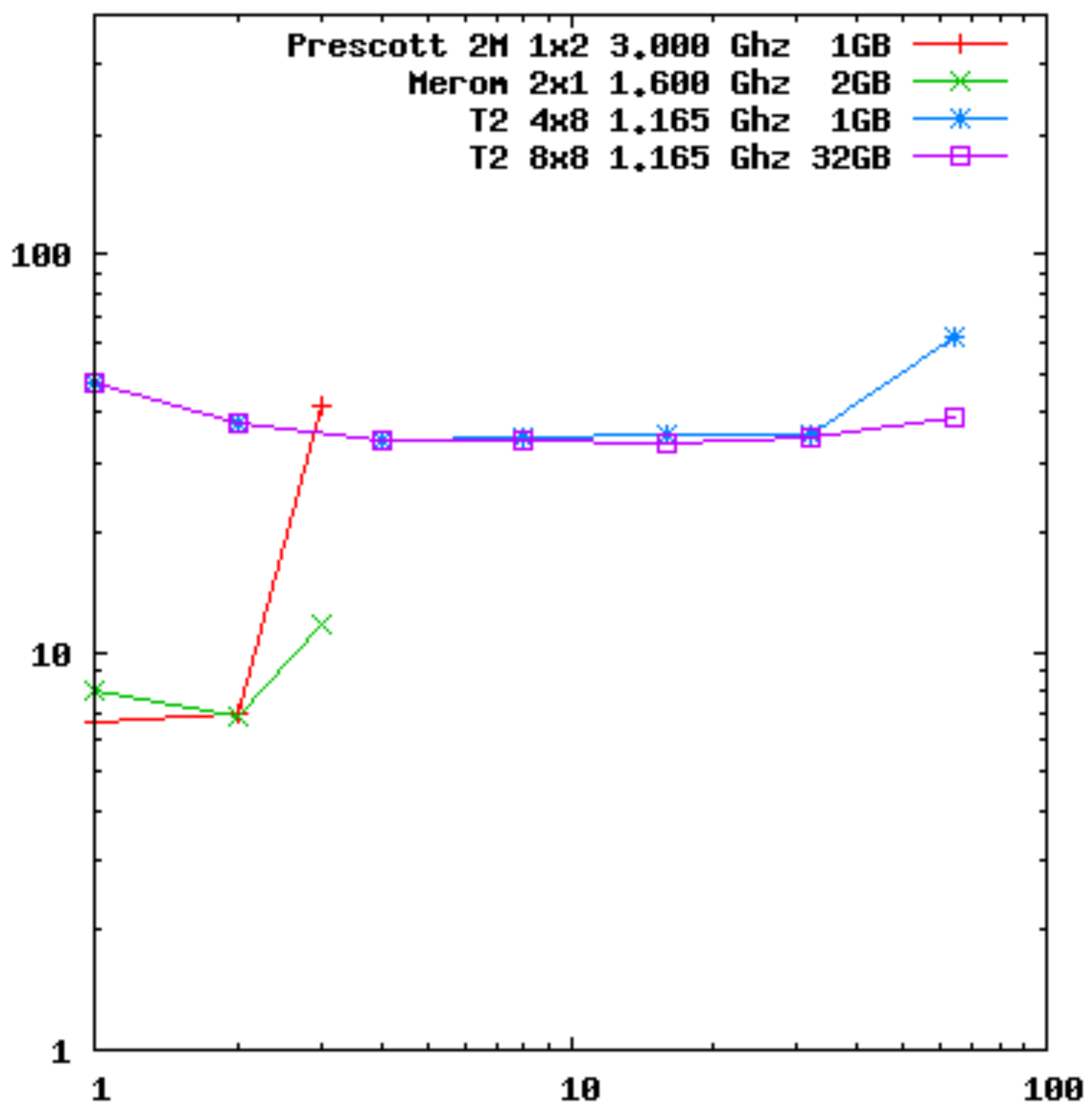
- Embarrassingly parallel benchmark.
- Use Intel processors as the baseline.
- Almost linear speedup until we run out of hardware threads.
- No point using more Haskell threads than hardware threads.

partree: runtime(s) *vs* number of threads



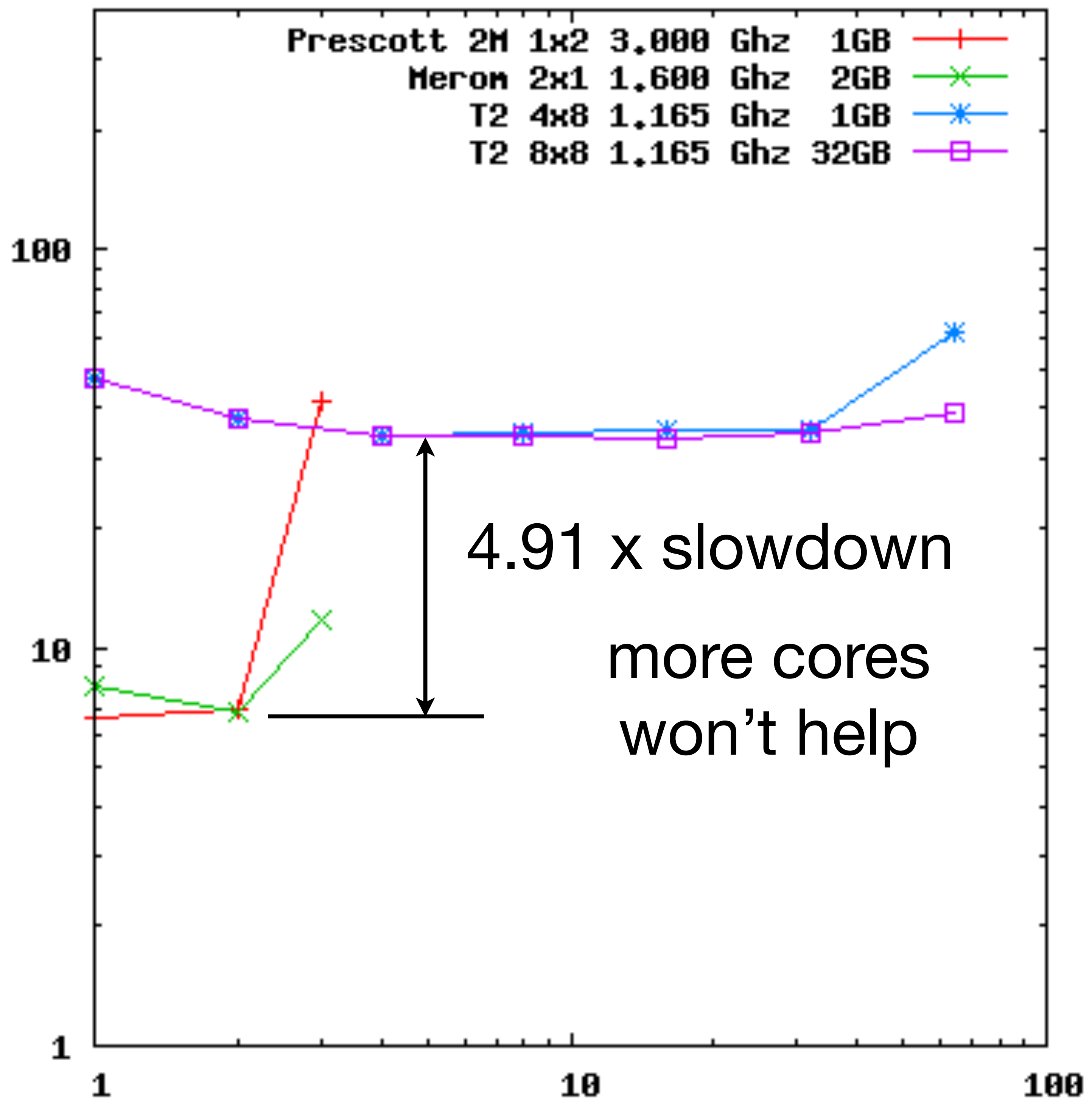
- Not very parallel.
- Tiny speedups on Intel.

partree: runtime(s) *vs* number of threads



- Not very parallel.
- Tiny speedups on Intel.
- No real speedup with more than 3 threads.
- Can't make full use of a whole T2 core.

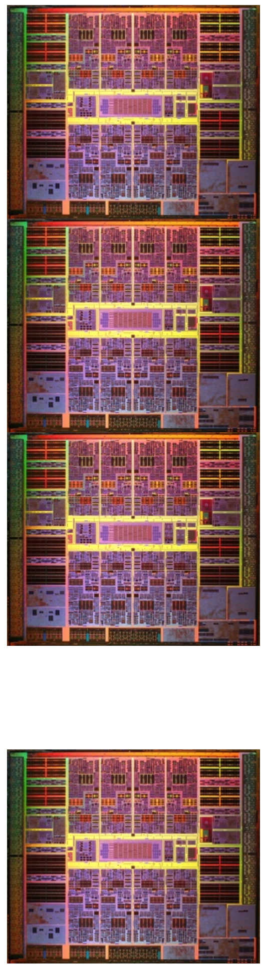
partree: runtime(s) *vs* number of threads



- Not very parallel.
- Tiny speedups on Intel.
- No real speedup with more than 3 threads.
- Can't make full use of a whole T2 core.

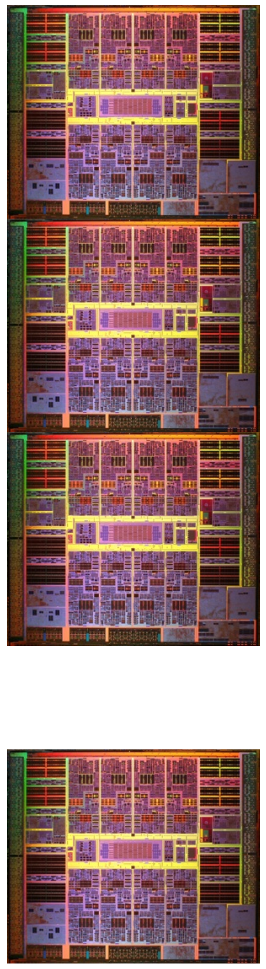
Benchmarking Summary

If you have less than 8 threads of work then stay home.



Benchmarking Summary

If you have less than 8 threads of work then stay home.



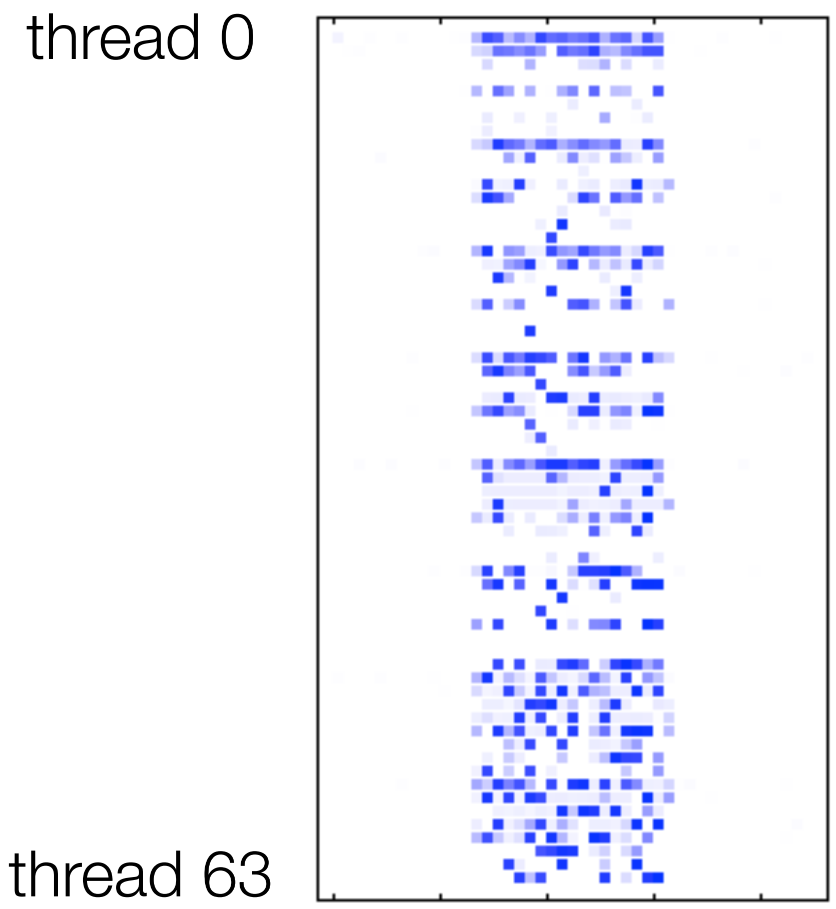
It’s a “throughput” machine.

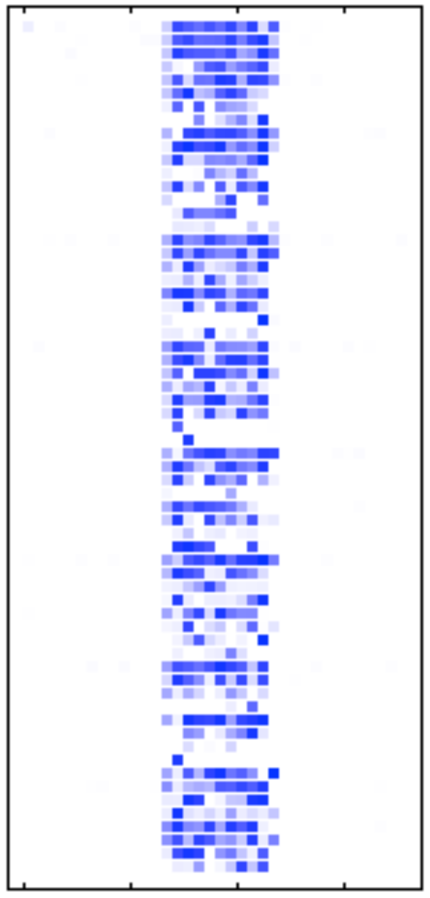
sumeuler: issue rate, data miss rate (Gig/s) *vs* time(s)

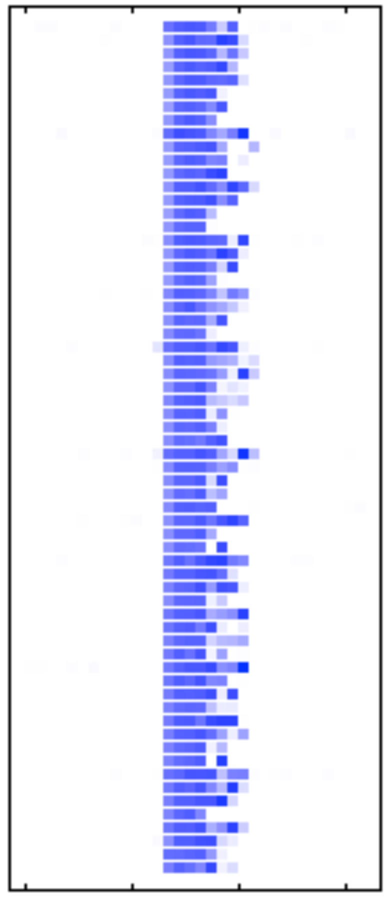
-N16

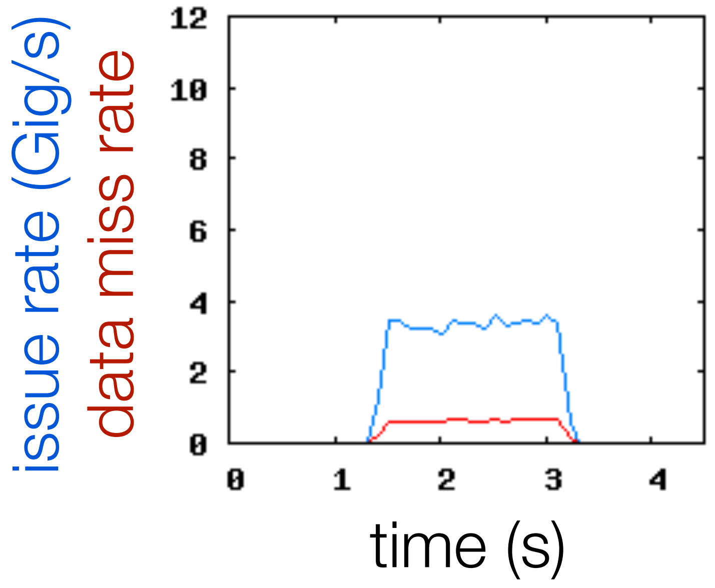
-N32

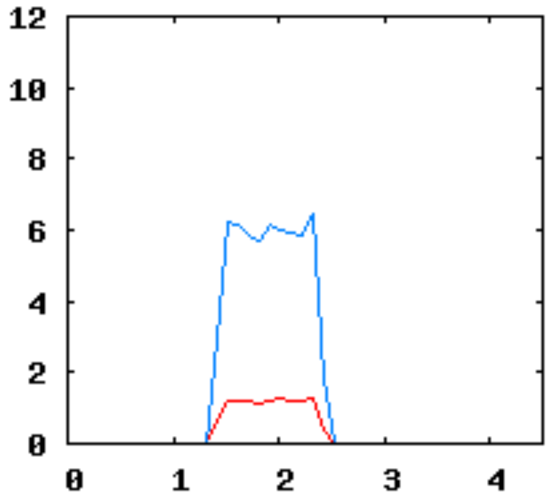
-N64

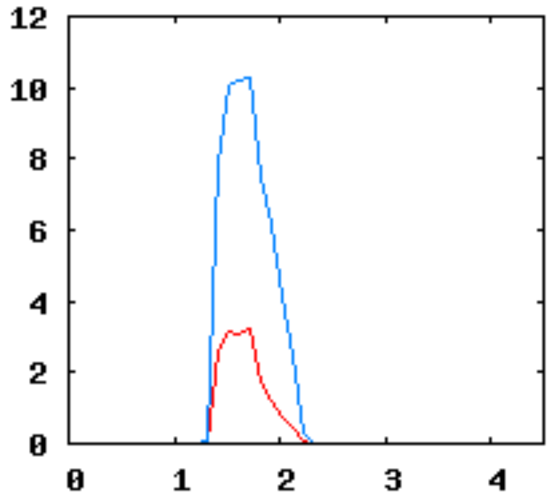




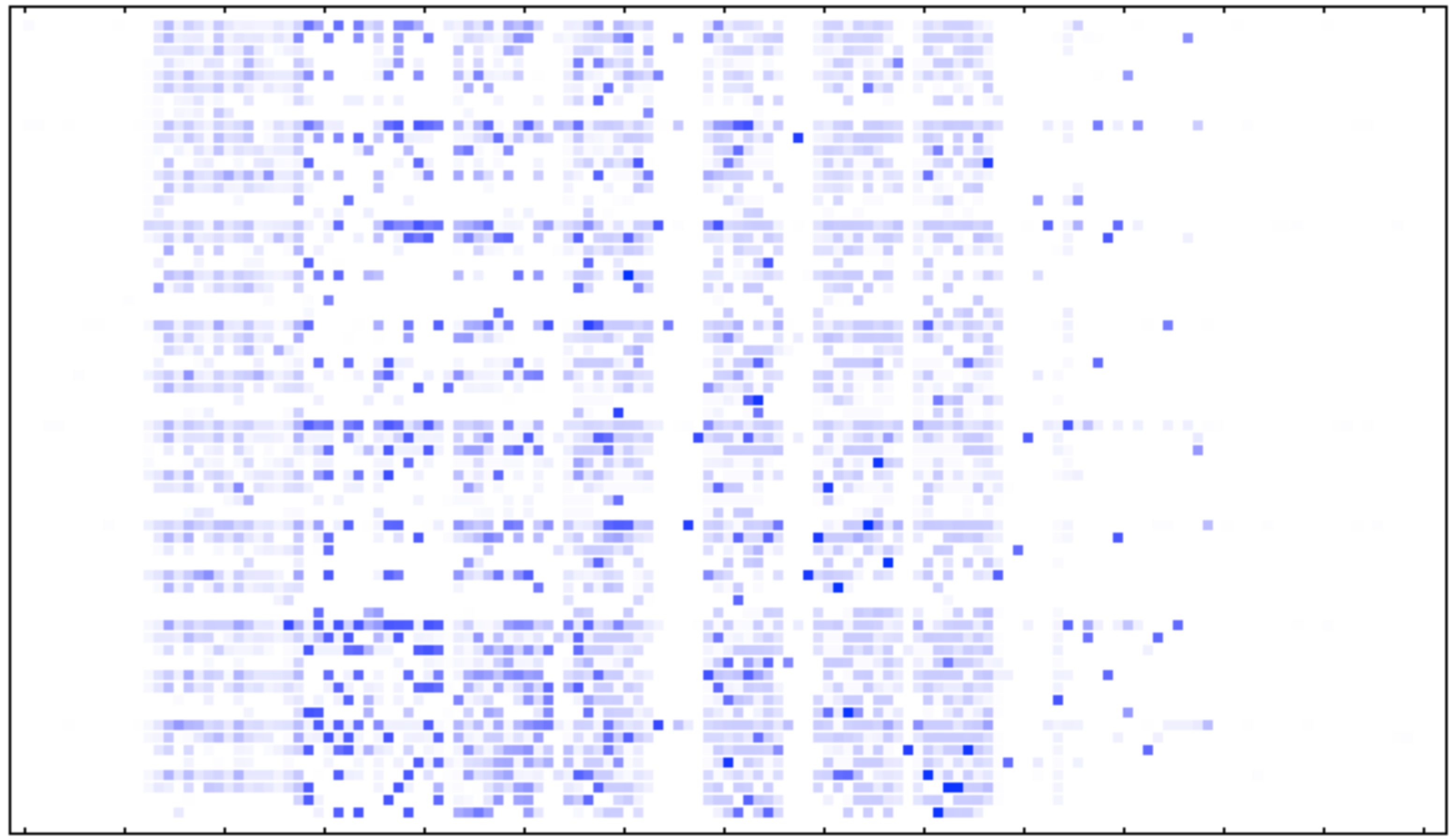


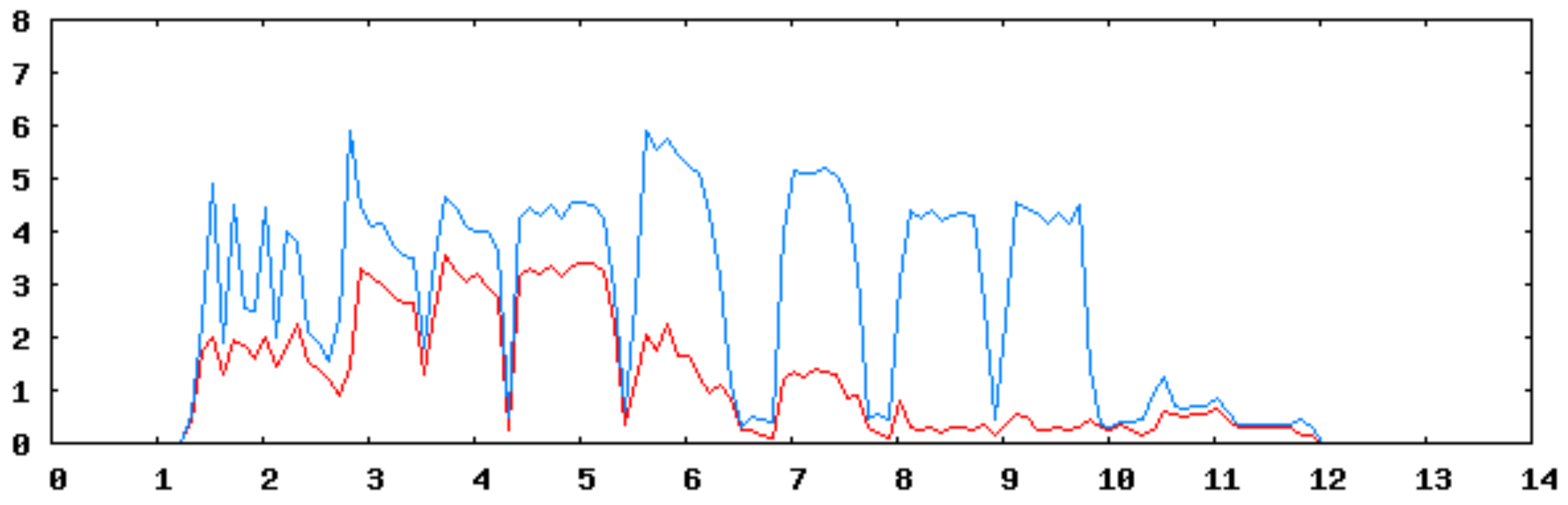






matmult: issue rate, data miss rate (Gig/s) *vs* time(s)





- Periods of high and low parallelism.
- Large variation run-to-run.
- Threads spend time blocked at join points?
- Can ThreadScope help debug this?

What next?

- We need more satisfying benchmarks.
- We haven’t had 64 hardware threads before.
- Use ThreadScope to determine why matmult is behaving badly.
- Some simple compile-time instruction reordering could help.
  - No out-of-order execution => pipeline stalls.
- Keep the build working!!

More info at:

`http://ghcsparc.blogspot.com`