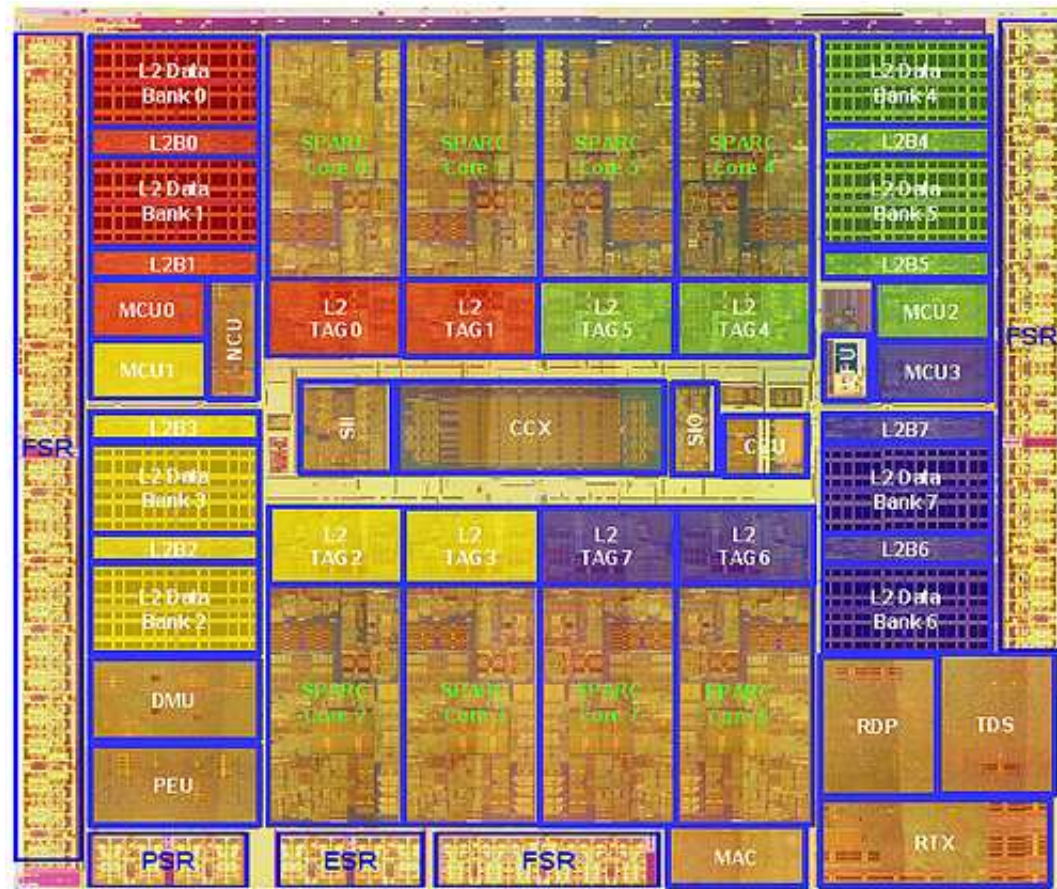


# Welcome to COMP8320 – Multicore Computing

UltraSPARC T2  
(Niagara-2)  
multicore chip layout

(courtesy of T. Okazaki, Flickr)



# Overview

- course contact
- review: instruction-level parallelism, memory hierarchy, cache memory, shared memory multiprocessor and programming model, OpenMP
- assumed knowledge and assessment
- intermission!
- discussion: your views on multicore
- what is multicore; four multicore designs
- advent of multicore (parallel programming decline and rebirth)
- the multicore challenge: why parallel programming is hard

credits: some material for this lecture is from CS194 course by Prof Kathy Yelick, UCB

# Course Contact

- course web site: <http://cs.anu.edu.au/student/comp8320>
- course coordinator & lecturer/tutor:  
Peter Strazdins, CSIT N219, 6125-5140, [comp8320@cs](mailto:comp8320@cs)
- discussion forum accessible by StReAMS:
- course schedule: 10 'modules'

## Review: Instruction Level Parallelism: Pipelining

- pipelining: achieve fast cycle times by breaking instr'n execution into  $k$  stages
  - if complete one instr'n per cycle. get  $\leq k$ -way parallelism
  - note: generally, the circuitry for each stage is independent
- e.g. ( $k = 5$ ): stages FI = Fetch Instr'n., DI = Decode Instr'n., FO = Fetch Operand, EX = Execute

Instr'n., WB = Write Back



- note: EX & WB stages may involve memory accesses (and may possibly stall the pipeline)

## Review: Instruction Level Parallelism: Superscalar

- multiple instruction issue (superscalar):

a small number ( $w$ ) of instr'ns (a group) are scheduled by the H/W to *execute together*

- groups must have an appropriate 'instruction mix'

e.g. UltraSPARC III ( $w = 4$ ):  $\left. \begin{array}{l} \leq 2 \text{ different floating point} \\ \leq 1 \text{ load / store ; } \leq 1 \text{ branch} \\ \leq 2 \text{ integer / logical} \end{array} \right\} \text{instr'ns per group}$

- have  $\leq w$ -way parallelism over *different* types of instr'ns

- generally requires:

- ◆ multiple ( $\geq w$ ) instr'n fetches; extra grouping stage (G) in the pipeline

- **problem**: amplifies all problems with pipelining

- **issue**: the instruction mix must be balanced for **maximum** performance!

- out-of-order execution

- hardware dynamically re-orders instructions to obtain better grouping and resolve dependencies (nice, but very complex and hence expensive!)

- e.g. swap the order of the **f**adds in

```
fmuld %f0,%f1,%f2 ; fadd %f2,%f4,%f4 ; ... ; fadd %f7,%f0,%f1
```



## Review: Cache Memory – Details

- idea: data that is “*currently most needed*” is brought into a (smaller) faster memory
- observation: memory accesses in *most* programs exhibit:
  - temporal locality: if access address  $X$ , likely to access  $X$  again soon
  - spatial locality: if access address  $X$ , likely to access  $X+1$  soon
    - ⇒ cache organized into lines (blocks) of  $B$  bytes ( $B = 2^b$ , e.g.  $4 \leq b \leq 9$ )
    - ✓ blocked memory accesses (faster) & less control info needed (per byte)
    - × redundant memory traffic if only ever use 1 byte per line
- if so, yields good cost-speed tradeoff
- problem: keeping consistency of data cache & main memory
  - when a `store` instruction is executed, the relevant line is updated 1st; when does main memory get updated?
    - ◆ 2 strategies: write-through and copy-back
- organization:  $S$ -way set associative caches ([O’H.&Bryant, fig 6.33])
  - middle bits of address  $X$  is used to select a set of  $S$  lines
  - typically  $S = 1, 2, 4, 6, 8$  (direct-mapped cache corresponds to  $S = 1$ )
  - reduces chance of conflicts by factor of  $S$

## Review: Shared Memory Multiprocessors

- a number of processors connected to a globally addressable memory
  - through a bus ( [Lyn&Snyder, fig 2.3]) or, better,
  - memory is organized into modules, all connected by an interconnect
- need caches! memory consistency problem is now exacerbated!
  - 2 strategies: invalidate and update
  - question: how *soon* do changes need to be propagated?
- consider programs (threads) on processors 0 and 1 attempting to acquire a lock
  - hardware must support this via some atomic instructions, e.g.

```
                                ! %o0 has address of the lock
                                ! 0xff is value for acquiring lock
    mov    0xff, %o1
loop:                                ! loop to acquire lock
    brnz  %o1, loop                ! exit if %o1 = 0 (lock value was 0)
    ldstub [%o0], %o1             ! atomic swap %o1 and value at lock
    ...                            ! safely update shared data structure
    stb   %g0, [%o0]              ! release lock
```

- for locking to work properly, any change to a memory location by 1 processor must be known at the next atomic instruction

## Review: the Shared Memory Programming Model

- the threaded programming model [Chap.&Jost&derPas, fig 2.1]
  - the team of threads executes a parallel region
  - for parallel speedup, each thread needs to be allocated to a different processor
  - in the low-level threads library (`pthread`s), only fork or join 1 thread at once
  - in OpenMP, conceptually like the above
    - ◆ however, threads are typically created at the start of program, and are allocated work upon each parallel region
- threads communicate via global variables in common memory sections (e.g. static data, heap); have private stacks for thread-local variables
  - see [Hughes&Hughes, fig 3.2]

- main synchronization mechanisms are locks and barriers

```
pthread_mutex_init(&mutex1, NULL);  
...  
pthread_mutex_lock(&mutex1); // involves a busy-wait loop  
/* mutually exclusive access to shared resource */  
pthread_mutex_unlock(&mutex1);  
...  
pthread_barrier_wait(&barrier); // wait until other threads reach the same point
```

## Overview of the OpenMP Programming Model

- idea is to add directives to ordinary C, C++ or Fortran code
- example: matrix-vector multiply  $y \leftarrow y + Ax$  ([Chap.&Jost&derPas, fig 3.9])

```
double A[N][N], x[N], y[n]; int i, j;
...
#pragma omp parallel for private(i,j)
for (i=0; i < N; i++) // each thread updates segment of y[]
    for (j=0; j < n; j++)
        y[i] += A[i][j] * x[j];
// alternately:
for (i=0; i < N; i++) {
    double s = y[i];
#pragma omp parallel for private(j) reduction(+:s)
    for (j=0; j < n; j++) // each thread computes a partial sum
        s += A[i][j] * x[j];
    y[i] = s;
}
```

- more generally, directives apply over regions `#pragma omp parallel { ... }`
- within a parallel region, a barrier may be inserted (`#pragma omp barrier`)
- mutual exclusion via `#pragma omp critical { ... }`
- for details, see LNL Tutorial or comp4300 lecture

## Proposed Course Assessment Scheme

- see the assessment web page
- Assignments: 50%, three, programming + report:

#	theme	weight	week out	week due
1	T2	20%	2	6
2	GPU	15%	6	8
3	SCC	15%	9	12

- Final Exam, 50%, will cover all the course,
- *this is an advanced course!* Assumes u/g course in computer architecture and concurrency! Also self-motivated study.
  - need strong technical ability to use state-of-the-art tools and languages (may be 'bleeding edge')
  - there are no real text books; multicore computing is too new!
    - ◆ see the references page
  - tutorial and project components will require you to read and understand research papers from the literature



## What is Multicore?

- (also known as chip multiprocessing, CMP): multiple shared memory processors ('cores' ) on a single chip
  - why? because we can! also because we must . . .
  - can run multiple applications (processes) in parallel
  - can run a single (threaded) application in parallel; **herein lies the challenge!**
- (large-scale) parallelism is now cheap, mainstream
- some multicore designs:
  - Intel Core 2 Duo ([Hughes&Hughes, fig 2-8]): shared L2\$; coherency needed on separate L1\$s
  - AMD Opteron ([Hughes&Hughes, fig 2-3]): separate L2\$; coherency needed on L2\$s (both within and across chips ('sockets'))
  - UltraSPARC T1 ([Hughes&Hughes, fig 2-5]): note crossbar for the L2\$ and multiple memory controllers; also uses hardware threading (why?)
  - Cell Broadband Engine ([Hughes&Hughes, fig 2-6]): heterogeneous cores (SPEs) of limited capability  $\Rightarrow$  greater programming issues!

Memory (data access) is a key consideration!



## Advent: End of Clock Scaling and Instruction-level Parallelism

- extrapolation of exponential power density increase 1985–2000 indicates we are at the limit!
  - 2000 Intel chip equivalent to a hotplate, would have  $\Rightarrow$  a rocket nozzle by 2010!
- dissipated power is given by:  $P \propto CV^2f \propto Cf^3$ ,  $V$  is the voltage,  $C$  is the chip's capacitance and  $f$  is the clock frequency
- (*ideal*) parallel performance for  $p$  cores is given by  $R = pf$ , but  $C \propto p$
- double  $p \Rightarrow$  double  $R$ , but also  $P$
- double  $p$ , halve  $f \Rightarrow$  maintain  $R$ , but quarter  $P$
- instruction level parallelism has reached its limits
  - instructions per cycle (IPC) of a 6-way superscalar not much better than a 2-way (Olukotun et al, ASPLOS, 1996)
  - by utilizing technique of hardware threading, can hide effects of cache misses
    - ◆ for each core, have a number of 'virtual CPUs', each with own register set
    - ◆ the core's control unit can flexibly select instructions ready to execute in each of these



## Software: The Multicore Challenge

- all computers are now parallel; all major vendors offer multicore
- will all programmers (software engineers) have to be (performance-aware) parallel programmers?
  - what programming model(s) should be used? (little current consensus)
  - or will compilers and libraries manage to hide it?
- how should multicore processors be designed?
  - what will be the “killer apps” for multicore computers?
    - ◆ web and online transactions servers well suited:
      - ★ potentially very many threads at once (under heavy load), seldom interacting but with frequent stalls
      - ★ very limited instruction level parallelism; high degree of dependencies; unpredictable branches
      - ★ speed can however come from trying to execute many threads (pseudo-) simultaneously!  
but for what about for personal use?
  - watch out for the memory wall!

## Software: Why Parallel Processing is Hard

- writing (correct and efficient) parallel programs is hard!
  - hard to expose enough parallelism; hard to debug!
- getting (close to ideal) speedup is hard! Overheads include:
  - communicating shared data (e.g. cache line invalidations and resulting reloads)
  - synchronization (barriers and locks)
  - need for redundant computations; balancing load evenly

Also, not all of the application may be parallelizable:

Amdahl's Law: given a fraction  $f$  of 'fast' computation, at rate  $R_f$ , and  $R_s$  being the 'slow' computation rate, the overall rate is:  $R = \left(\frac{1-f}{R_s} + \frac{f}{R_f}\right)^{-1}$

- interpreted for parallel execution with  $p$  processors:
  - $f$  is the fraction of non-serial computation, which (ideally) executes at the rate  $R_f = pR_s$
- e.g. with  $f = 0.9$ ,  $R = 10R_s$  at  $p = \infty$  !
- counterargument:  $1 - f$  is not fixed; but decreases with the data size  $N$   
e.g.  $f \propto N^{-\frac{1}{2}}$