CS2101: Foundations of High-performance Computing

Marc Moreno Maza

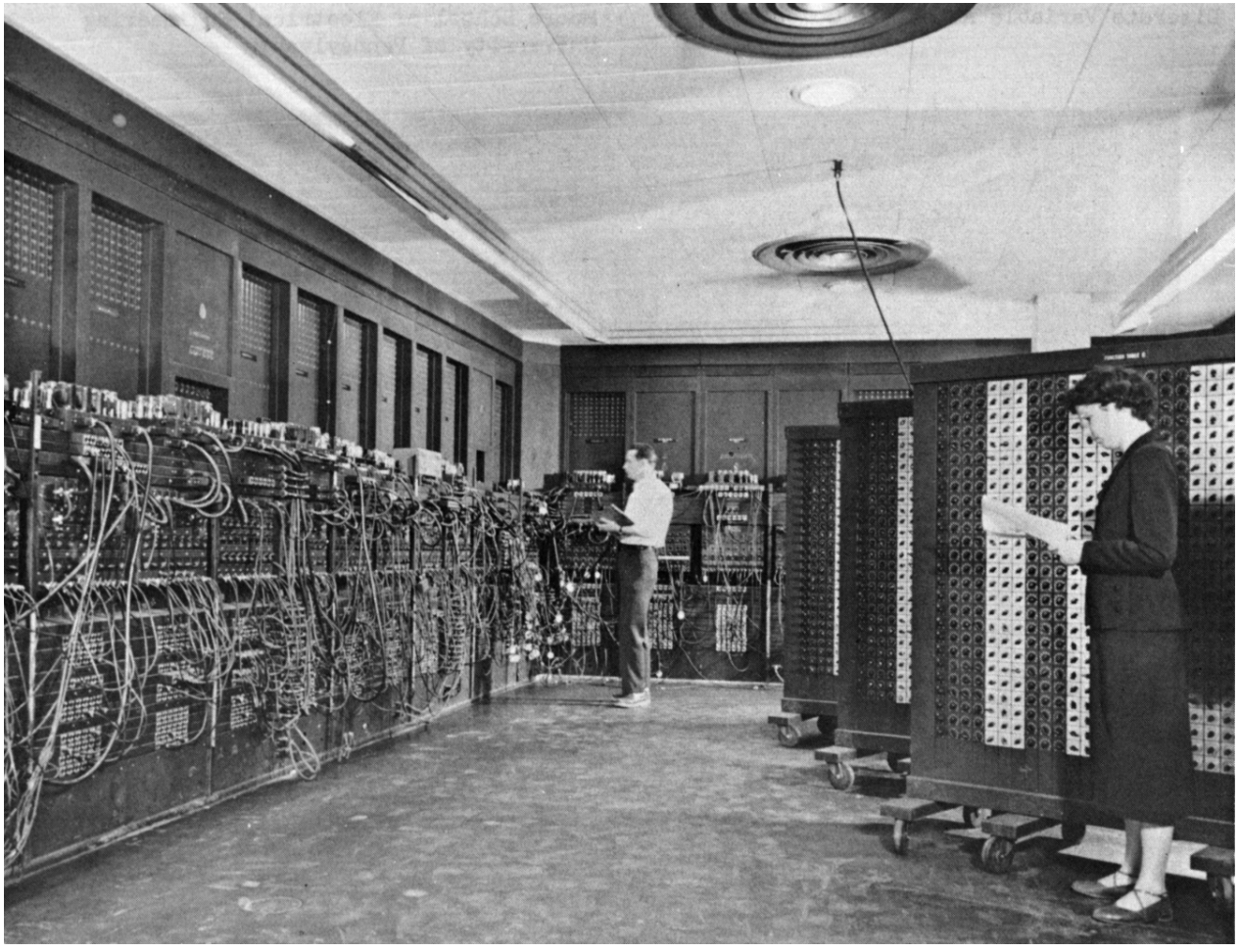
University of Western Ontario, London, Ontario (Canada)

CS2101

Plan

1. Hardware Acceleration Technologies
2. Software Performance Engineering
3. A Case Study: Matrix Multiplication
4. Multicore Programming
5. CS2101 Course Outline

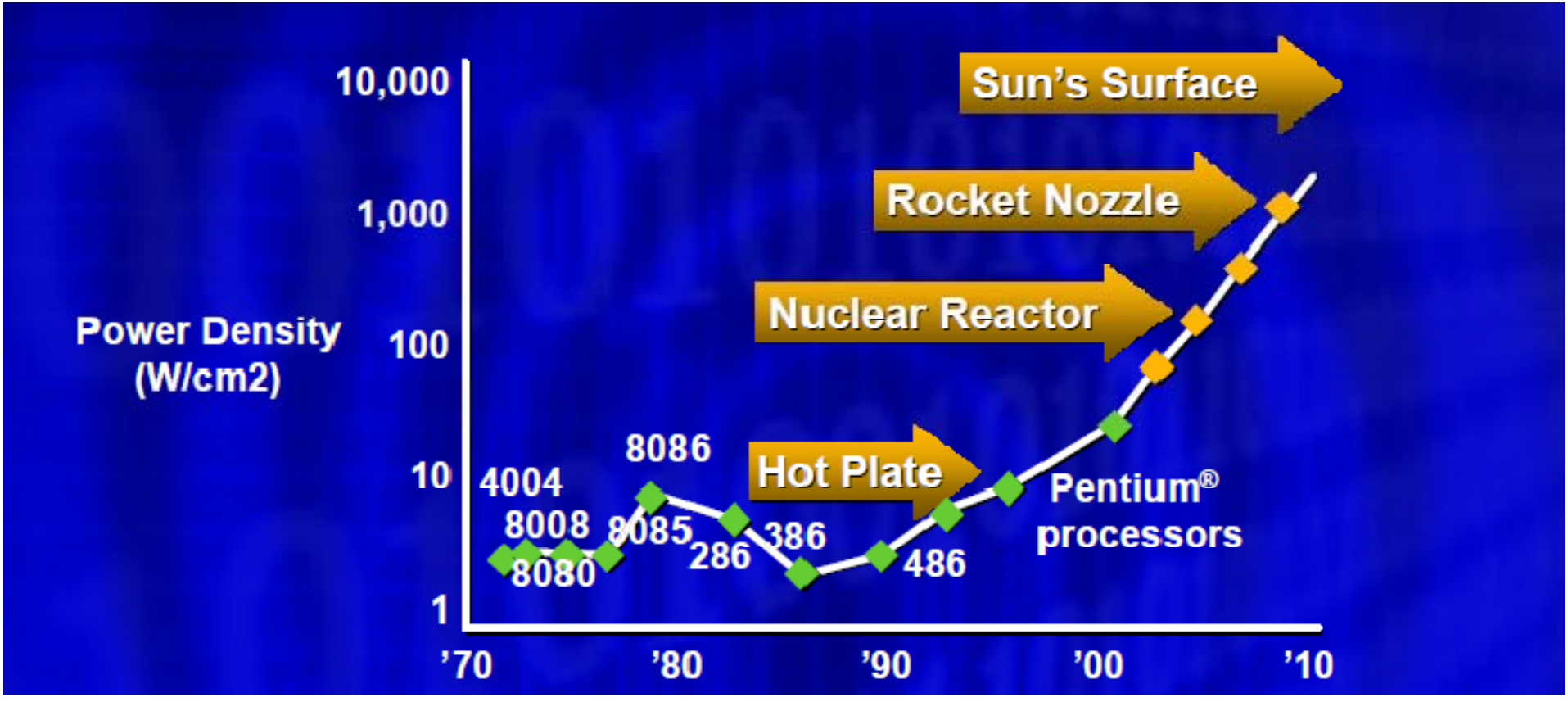
Plan



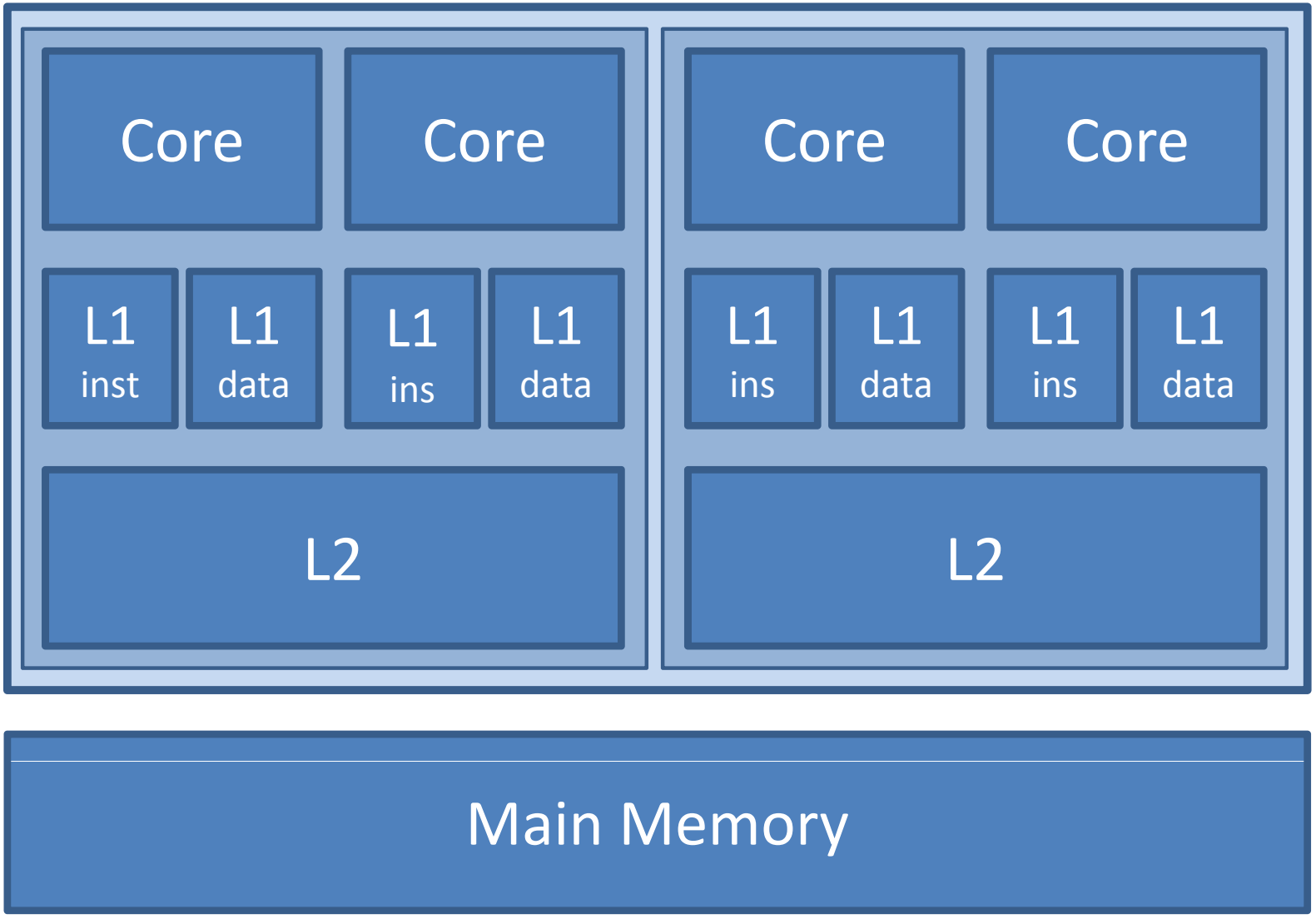
Electronic Numerical Integrator And Computer (ENIAC). The first general-purpose, electronic computer. It was a Turing-complete, digital computer capable of being reprogrammed and was running at 5,000 cycles per second for operations on the 10-digit numbers.

The IBM Personal Computer, commonly known as the IBM PC (Introduced on August 12, 1981).

The Pentium Family.


Power Density
(W/cm2)
10,000
1,000
100
10
1
Sun's Surface
Rocket Nozzle
Nuclear Reactor
Hot Plate
Pentium®
processors
4004
8008
8080
8085
8086
286
386
486
'70
'80
'90
'00
'10

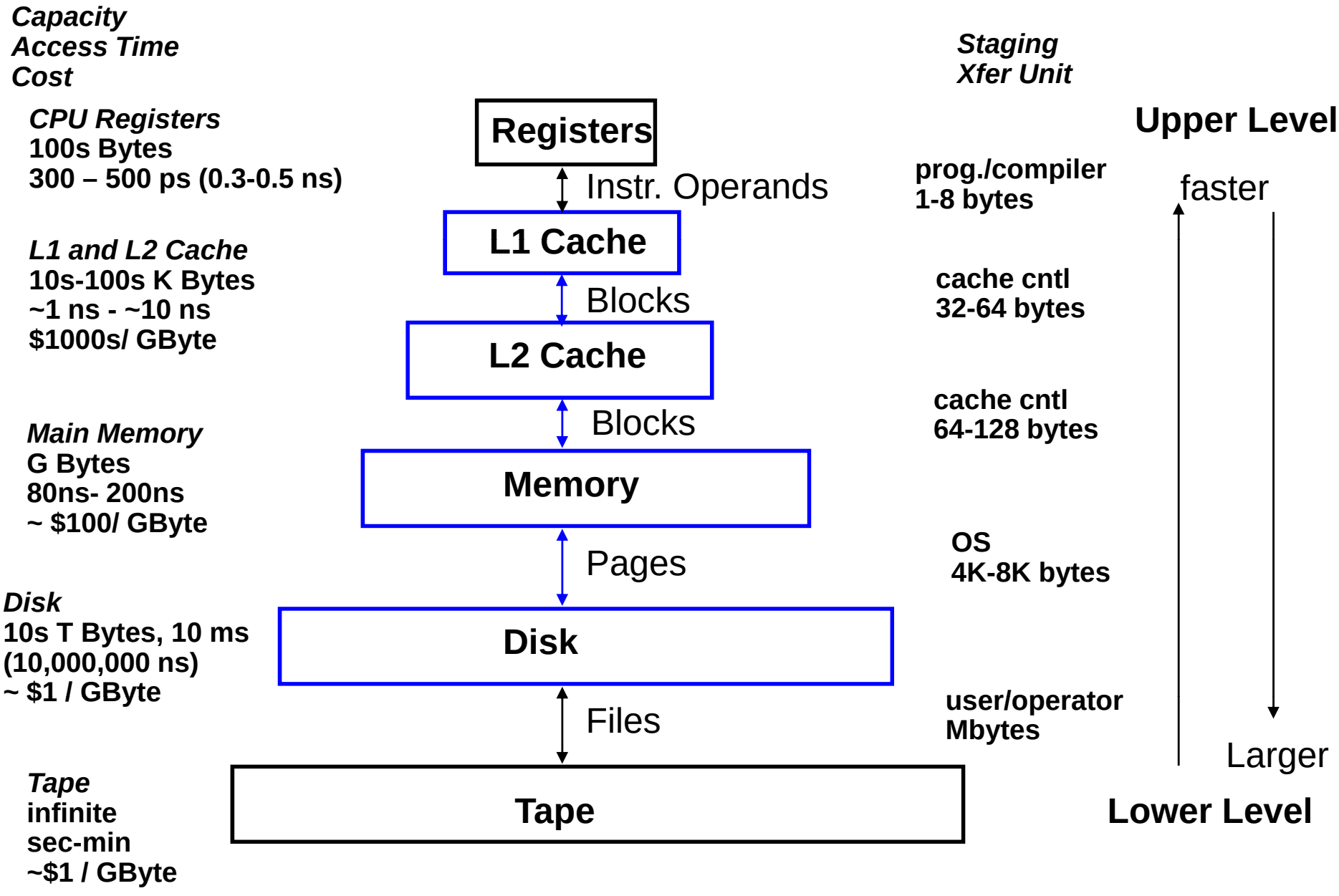

INTEL©'05 Q6600
INTEL® CORE™2 QUAD
SLACR MALAY
2.40GHZ/8M/1066/05A
L738B526


Core
Core
Core
Core
L1
inst
L1
data
L1
ins
L1
data
L1
ins
L1
data
L1
ins
L1
data
L2
L2
Main Memory


94V-0 MADE IN CHINA
NVIDIA
L7118760 0746A2
S TAIWAN
NF2043.00P
G92-270-A2
RoHS
COMPLIANT

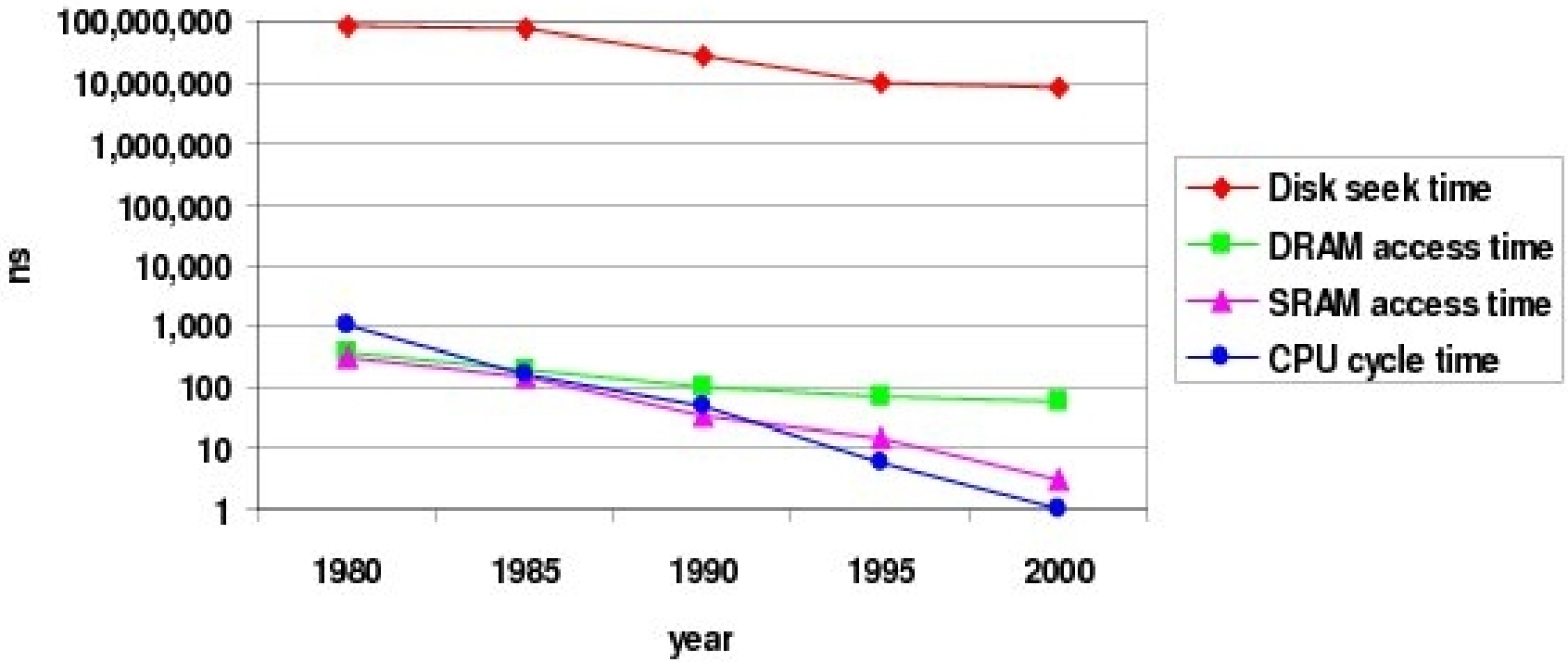
L1 Data Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L2 Cache			
Size	Line Size	Latency	Associativty
6 MB	64 bytes	14 cycles	24-way

Typical cache specifications of a multicore in 2008.


Capacity
Access Time
Cost
CPU Registers
100s Bytes
300 – 500 ps (0.3-0.5 ns)
L1 and L2 Cache
10s-100s K Bytes
~1 ns - ~10 ns
$1000s/ GByte
Main Memory
G Bytes
80ns- 200ns
~ $100/ GByte
Disk
10s T Bytes, 10 ms
(10,000,000 ns)
~ $1 / GByte
Tape
infinite
sec-min
~$1 / GByte
Registers
Instr. Operands
L1 Cache
Blocks
L2 Cache
Blocks
Memory
Pages
Disk
Files
Tape
Staging
Xfer Unit
prog./compiler
1-8 bytes
cache cntl
32-64 bytes
cache cntl
64-128 bytes
OS
4K-8K bytes
user/operator
Mbytes
Upper Level
faster
Larger
Lower Level

The CPU-Memory Gap

The increasing gap between DRAM, disk, and CPU speeds.



Once uopn a time, every thing was slow in a computer . . .

Plan

Why is Performance Important?

- Acceptable response time (Anti-lock break system, Mpeg decoder, Google Search, etc.)
- Ability to scale (from hundred to millions of users/documents/data)
- Use less power / resource (viability of cell phones dictated by battery life, etc.)

Improving Performance is Hard

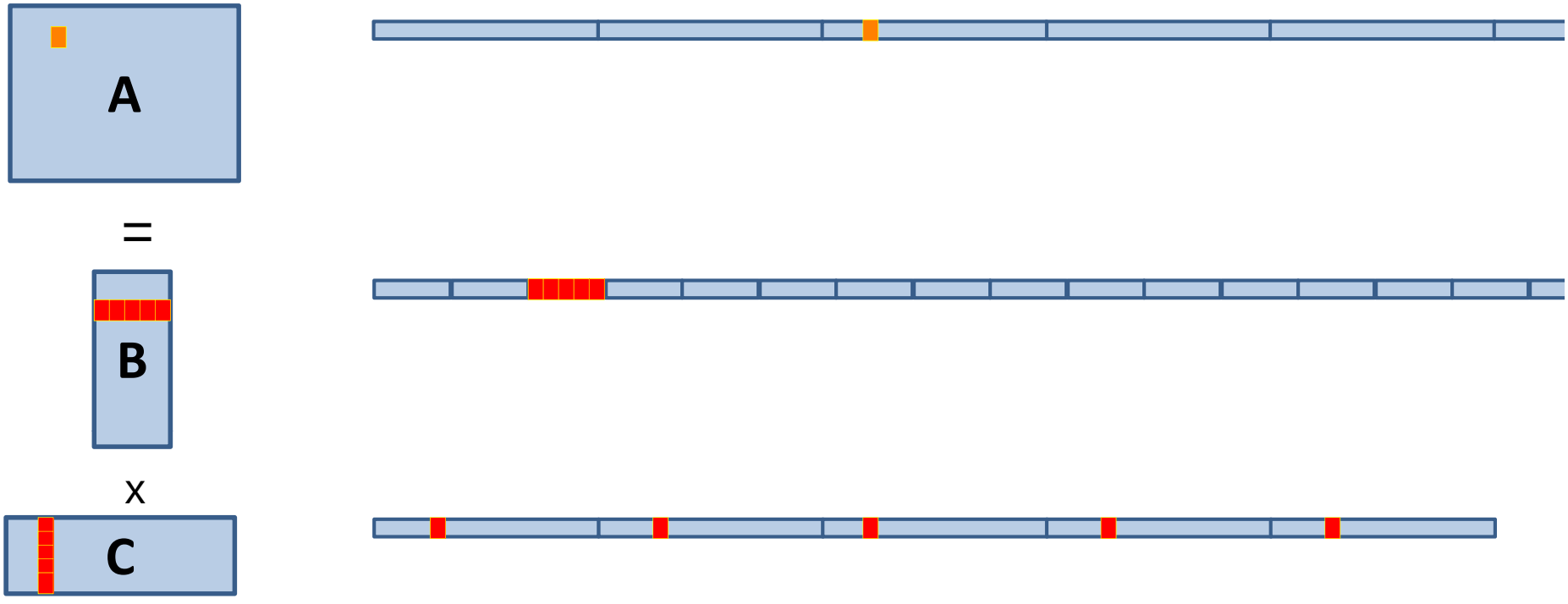
- Knowing that there is a performance problem: complexity estimates, performance analysis software tools, read the generated assembly code, scalability testing, comparisons to similar programs, experience and curiosity!
- Establishing the leading cause of the problem: examine the algorithm, the data structures, the data layout; understand the programming environment and architecture.
- Eliminating the performance problem: (Re-)design the algorithm, data structures and data layout, write programs *close to the metal* (C/C++), adhere to software engineering principles (simplicity, modularity, portability)
- Golden rule: Be reactive, not proactive!

Plan

A typical matrix multiplication C code

```
#define IND(A, x, y, d) A[(x)*(d)+(y)]
uint64_t testMM(const int x, const int y, const int z)
{
  double *A; double *B; double *C; double *Cx;
        long started, ended;
        float timeTaken;
        int i, j, k;
        srand(getSeed());
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i++)
          for (j = 0; j < y; j++)
             for (k = 0; k < z; k++)
                    //  A[i][j] += B[i][k] + C[k][j];
              IND(A,i,j,y) += IND(B,i,k,z) * IND(C,k,j,z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
  return timeTaken;
}
```

Issues with matrix representation

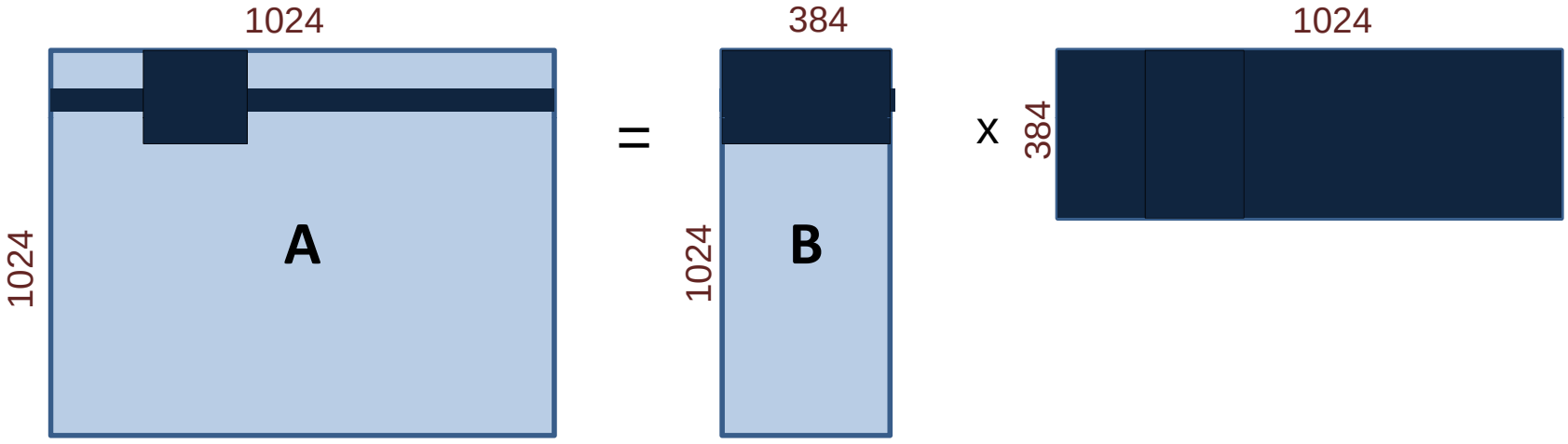


- Contiguous accesses are better:
  - Data fetch as cache line (Core 2 Duo 64 byte L2 Cache line)
  - With contiguous data, a single cache fetch supports 8 reads of doubles.
  - Transposing the matrix `C` should reduce L1 cache misses!

Transposing for optimizing spatial locality

```
float testMM(const int x, const int y, const int z)
{
  double *A; double *B; double *C; double *Cx;
        long started, ended; float timeTaken; int i, j, k;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        Cx = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for(j =0; j < y; j++)
          for(k=0; k < z; k++)
            IND(Cx,j,k,z) = IND(C, k, j, y);
        for (i = 0; i < x; i++)
          for (j = 0; j < y; j++)
             for (k = 0; k < z; k++)
               IND(A, i, j, y) += IND(B, i, k, z) *IND(Cx, j, k, z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
  return timeTaken;
}
```

Issues with data reuse



- Naive calculation of a row of A, so computing 1024 coefficients: 1024 accesses in A, 384 in B and $1024 \times 384 = 393,216$ in C. Total $= 394,524$.
- Computing a 32×32-block of A, so computing again 1024 coefficients: 1024 accesses in A, 384×32 in B and 32×384 in C. Total $= 25,600$.
- The iteration space is traversed so as to reduce memory accesses.

Blocking for optimizing temporal locality

```
float testMM(const int x, const int y, const int z)
{
        double *A; double *B; double *C; double *Cx;
        long started, ended; float timeTaken; int i, j, k, i0, j0, k0;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i += BLOCK_X)
          for (j = 0; j < y; j += BLOCK_Y)
            for (k = 0; k < z; k += BLOCK_Z)
              for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
                for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
                   for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
            IND(A,i0,j0,y) += IND(B,i0,k0,z) * IND(C,k0,j0,z);
         ended = example_get_time();
         timeTaken = (ended - started)/1.f;
   return timeTaken;
}
```

Transposing and blocking for optimizing data locality

```
float testMM(const int x, const int y, const int z)
{
        double *A; double *B; double *C; double *Cx;
        long started, ended; float timeTaken; int i, j, k, i0, j0, k0;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i += BLOCK_X)
          for (j = 0; j < y; j += BLOCK_Y)
            for (k = 0; k < z; k += BLOCK_Z)
              for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
                for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
                   for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
              IND(A,i0,j0,y) += IND(B,i0,k0,z) * IND(C,j0,k0,z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
        return timeTaken;
}
```

Experimental results

Computing the product of two $n \times n$ matrices on my laptop (Core2 Duo CPU P8600 @ 2.40GHz, L1 cache of 3072 KB, 4 GBytes of RAM)

n	naive	transposed	speedup	64×64-tiled	speedup	t. & t.	speedup
128	7	3		7		2	
256	26	43		155		23	
512	1805	265	6.81	1928	0.936	187	9.65
1024	24723	3730	6.62	14020	1.76	1490	16.59
2048	271446	29767	9.11	112298	2.41	11960	22.69
4096	2344594	238453	**9.83**	1009445	**2.32**	101264	**23.15**

Timings are in milliseconds.

The cache-oblivious multiplication (more on this later) runs within 12978 and 106758 for $n = 2048$ and $n = 4096$ respectively.

Plan

From Cilk to Cilk++ and Cilk Plus

- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo.
- Besides being used for research and teaching, `Cilk` was the system used to code the three world-class chess programs: Tech, Socrates, and Cilkchess.
- Over the years, the implementations of `Cilk` have run on computers ranging from networks of Linux laptops to an 1824-nodes Intel Paragon.
- From 2007 to 2009 Cilk has lead to `Cilk++`, developed by Cilk Arts, an MIT spin-off, which was acquired by Intel in July 2009 and became Cilk Plus, see `http://www.cilk.com/`
- `Cilk++` can be freely downloaded at `http://software.intel.com/en-us/articles/download-intel-ci`
- `Cilk` is still developed at MIT `http://supertech.csail.mit.edu/cilk/`

Cilk++ (and Cilk Plus)

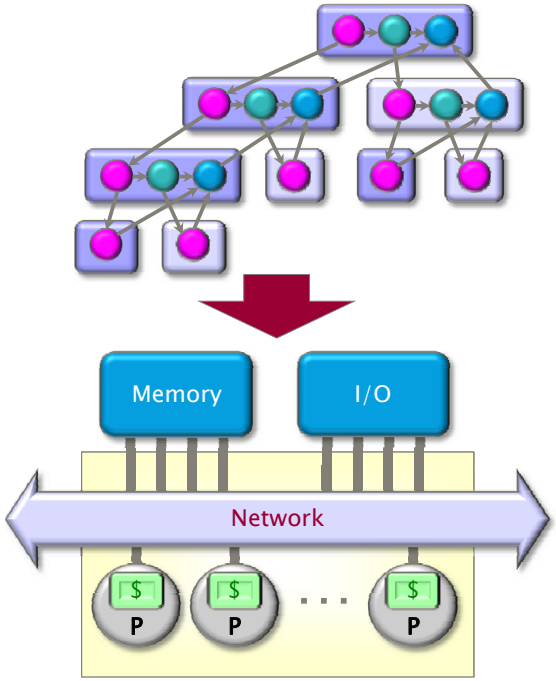
- `Cilk++` (resp. `Cilk`) is a small set of linguistic extensions to `C++` (resp. `C`) supporting fork-join parallelism
- Both `Cilk` and `Cilk++` feature a provably efficient work-stealing scheduler.
- `Cilk++` provides a hyperobject library for parallelizing code with global variables and performing reduction for data aggregation.
- `Cilk++` includes the `Cilkscreen` race detector and the `Cilkview` performance analyzer.

Nested Parallelism in Cilk ++

```
int fib(int n)
{
    if (n < 2) return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x+y;
}
```

- The named child function `cilk_spawn fib(n-1)` may execute in parallel with its parent
- Cilk++ keywords `cilk_spawn` and `cilk_sync` grant permissions for parallel execution. They do not command parallel execution.

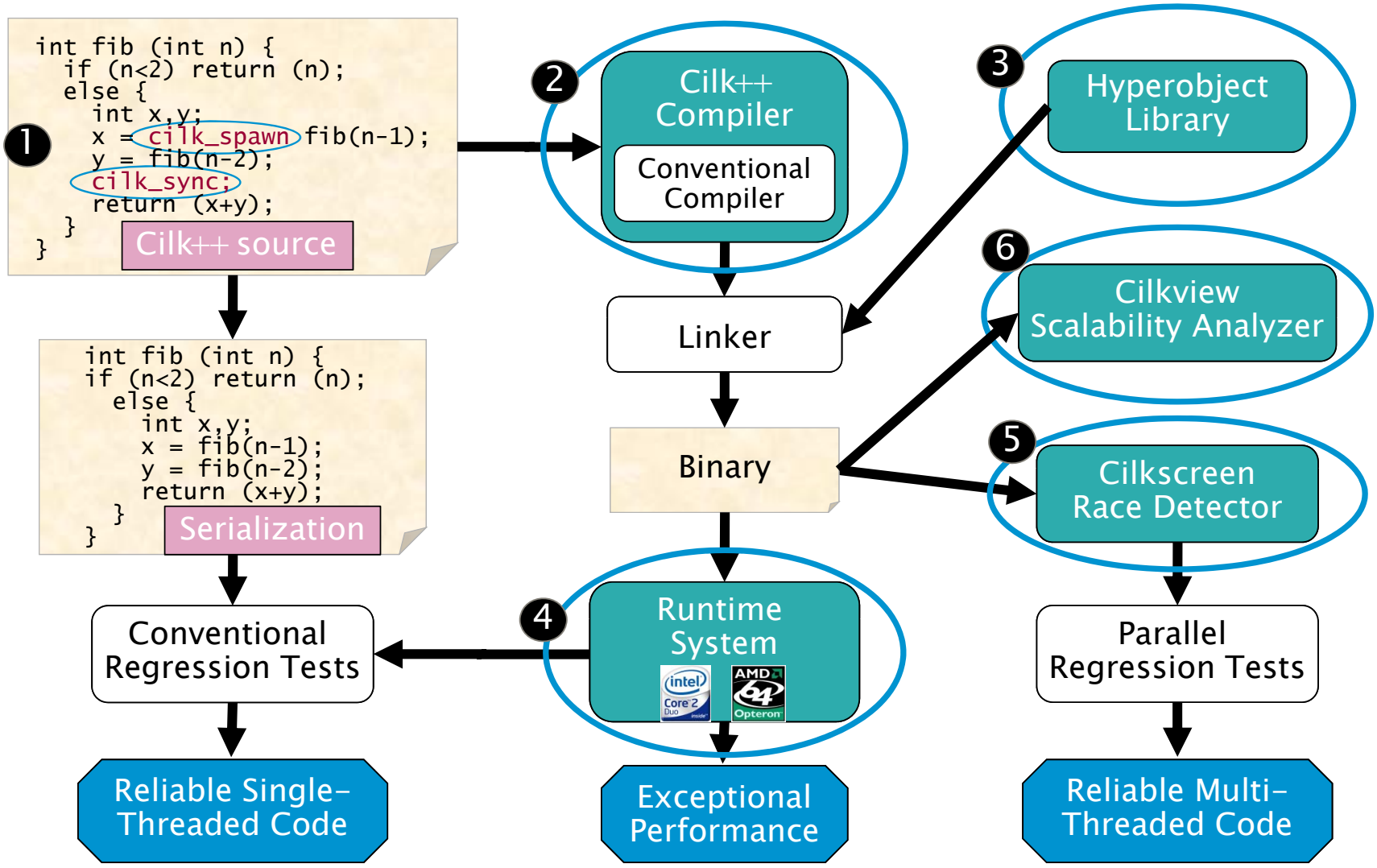
Scheduling



A **scheduler**'s job is to map a computation to particular processors. Such a mapping is called a **schedule**.

- If decisions are made at runtime, the scheduler is *online*, otherwise, it is *offline*
- `Cilk++`'s scheduler maps strands onto processors dynamically at runtime.

The Cilk++ Platform



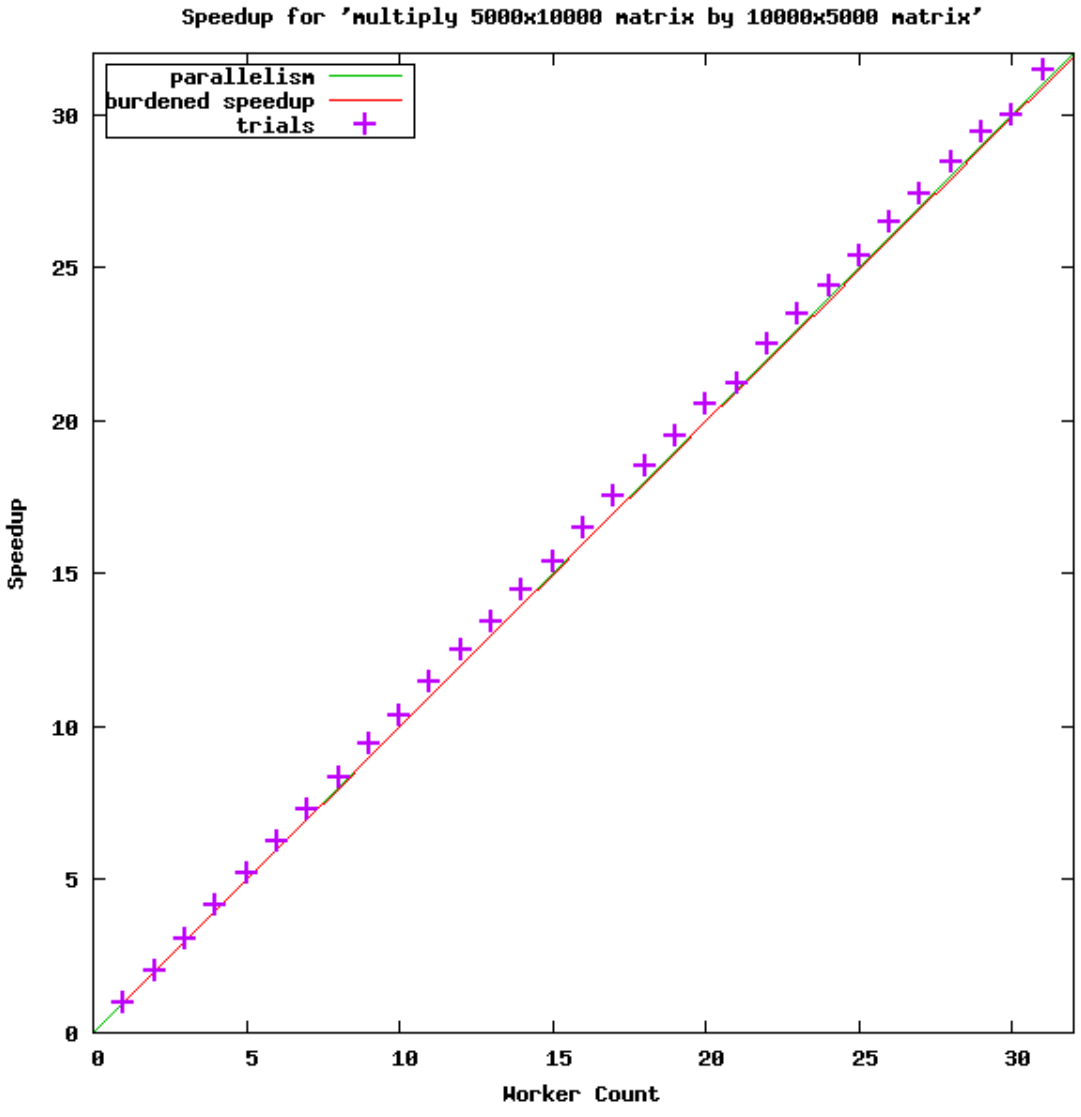
Benchmarks for the parallel version of the cache-oblivious mm

Multiplying a 4000x8000 matrix by a 8000x4000 matrix

- on 32 cores = 8 sockets x 4 cores (Quad Core AMD Opteron 8354) per socket.
- The 32 cores share a L3 32-way set-associative cache of 2 Mbytes.

#core	Elision (s)	Parallel (s)	speedup
8	420.906	51.365	8.19
16	432.419	25.845	16.73
24	413.681	17.361	23.83
32	389.300	13.051	29.83

So does the (tuned) cache-oblivious matrix multiplication



Plan

Course Topics

Week 1: Review of UNIX basics (command lines, editors) and C basics (basic types, flow of control, expressions, functions)

Week 2: Arrays and pointers in C

Week 3: UIX Fundamentals (permissions, regular expressions, shell programming, Makefiles)

Week 4: Issues with performance on single-core machines

Week 5: Data locality

Week 6: Multi-core architectures

Weeks 7-8: Multicore programming

Week 9-10: Multithreaded parallelism and performance measures

Week 11: Analysis of multithreaded algorithms

Weeks 12: Issues with code parallelization and data locality

About this course

- Prerequisites: no CS courses, but familiarity with a programming language. Knowledge of linear algebra, linear recurrences is assumed.
- Objectives: introduce students to *performance software enginnering*.
- Methods: build a strong knowledge in C/UNIX, then study multicore programming in Cilk.
- We will cover a large of materials and we will have tutorial every week.