CS2101a – Foundations of Programming for High Performance Computing

Xiaohui Chen & Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS2101

Plan

1. Course Overview
2. Hardware Acceleration Technologies
3. High-performance Computing
4. A Case Study: Matrix Multiplication

Plan

Course description (1/2)

In two sentences:

- This course is an introduction to *parallel computing* and its *applications in science*.
- The emphasis is on the *usage* of modern parallel computer architectures and concurrency platforms rather than the design of parallel algorithms and the optimization of computer programs.

The audience:

- The targeted audience is undergraduate students who are not engaged in a computer science program,
- but who want to be exposed to the principles of HPC and take advantage of them in their field of study.

Course description (1/2)

Objectives:

- Students will be introduced to the ideas and techniques that underline the usage of multicore architectures, GPUs and clusters.
- They will be presented with software that are commonly used in scientific computing, namely Matlab and Julia.
- They will study *fundamental parallel algorithms* (mainly from an experimental viewpoint) and assemble them in course projects within Julia.

Multicore architectures, GPUs and clusters (1/2)

Once upon a time (well, 10 years ago)

- All personal computers were essentially identical and using single-core processors.
- Most programmers needed not to know how computers worked.

But today:

- Personal computers cover several types (laptops, tablets, smart phones, workstations)
- Moreover, they all are parallel machines thanks to hardware accelerators (multicore architectures, GPUs) and may be tightly connected (thus forming clusters)

Multicore architectures, GPUs and clusters

Multicore architectures

Typically a multicore processor consists of 2, 4, 6, 8, 12, 16, 24, 32, 48 or 64 cores sharing memory and capable of working on either the same program or different programs.

GPUs

Graphics Processor Units (GPUs) consists of much more cores (typically 1024) sharing memory and (up to some details) executing simultaneously the same program.

Clusters

A typical cluster consists of a few workstations, not sharing memory, but capable of running the same program, by exchanging messages through the network.

Matlab

```
>> A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
A =
 16  3  2 13
  5 10 11  8
  9  6  7 12
  4 15 14  1

>> A(2,3)
ans =
 11
```

- MATLAB (matrix laboratory) is a numerical computing environment
- It offers a very high level programming language, which is interpreted by a *read-eval-print* interactive loop
- MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms and provides support for parallel computing
- This is a commercial software developed by MathWorks which is very popular among engineers

Julia

```
julia> sqrt(-1)
NaN

julia> sqrt(-1 + 0im)
0.0 + 1.0im
```

- Julia is another high-level dynamic programming language designed to support numerical computing
- Its designers emphasize two aspects: high performance and expressiveness
- This is an academic project which started at MIT around Alan Edelman
- Julia is free and open source

Parallel algorithms: challenges (1/3)

The Euclidean Algorithm

```
function gcd(a, b)
    while b <> 0
        t := b
        b := a mod t
        a := t
    return a
```

There is no (reasonably efficient) parallel version of this fundamental algorithm.

Parallel algorithms: challenges (2/3)

Naive matrix multiplication

The following code fragment multiplies the matrix A with the matrix B then writes the result to C

```
for i in 1..m  do
    for j in 1..p do
        for k in 1..n do
            C[i,j] := C[i,j] + A[i,k] * B[k, j]
```

Writing a reasonably efficient parallel code for the same task is hard.

A first attempt

```
for i in 1..m  do
    for j in 1..p do
        paralle_for k in 1..n do
            C[i,j] := C[i,j] + A[i,k] * B[k, j]
```

Is an incorrect program. Why?

Parallel algorithms: challenges (3/3)

A second attempt

```
paralle_for i in 1..m  do
    paralle_for j in 1..p do
        for k in 1..n do
            C[i,j] := C[i,j] + A[i,k] * B[k, j]
```

As we shall see, the above is a correct program, but very inefficient.

A third attempt

Below, for simplicity, we assume $m = n = p$ and consider a positive integer B dividing n.

```
parallel_for x in 0..(n/B -1) do
   parallel_for y in 0..(n/B -1) do
      for z in 0..(n/B -1) do
         for i in 1.B  do
            for k in 1..B do
                for k in 1..B do
                   (a,b,c) := (x*B+i, y*B+j, z*B+k)
                   C[a,b] := C[a,b] + A[a,c] * B[c, b]
```

As we shall see, this *block-wise* version of matrix multiplication, for a well chosen b is practically optimal.

Course Topics (1/2)

Week 1: Course presentation and orientation

Week 2: Overview of parallel computing: architectures, programming schemes, challenges and applications

Week 3: Julia

Week 4: Matlab

Week 5: Memory organization in modern processors: cache memories, cache misses, spatial and temporal data locality, optimizing code for data locality

Week 6: Multicore architectures: memory consistency, true/false sharing, advantages and limitations of multicore architectures, task graphs, work and span, scheduling, performance metrics (speedup, efficiency, scalability), parallelization overheads, optimizing code for parallelism

Weeks 7: Prefix sum: this fundamental operation has many applications (in particular in data processing) and will serve the notions on parallelism introduced in the previous weeks

Course Topics (2/2)

Week 8: Matrix Multiplication: this other fundamental operation is behind many applications in scientific computing and will illustrate the notions related to parallelism and memory organization

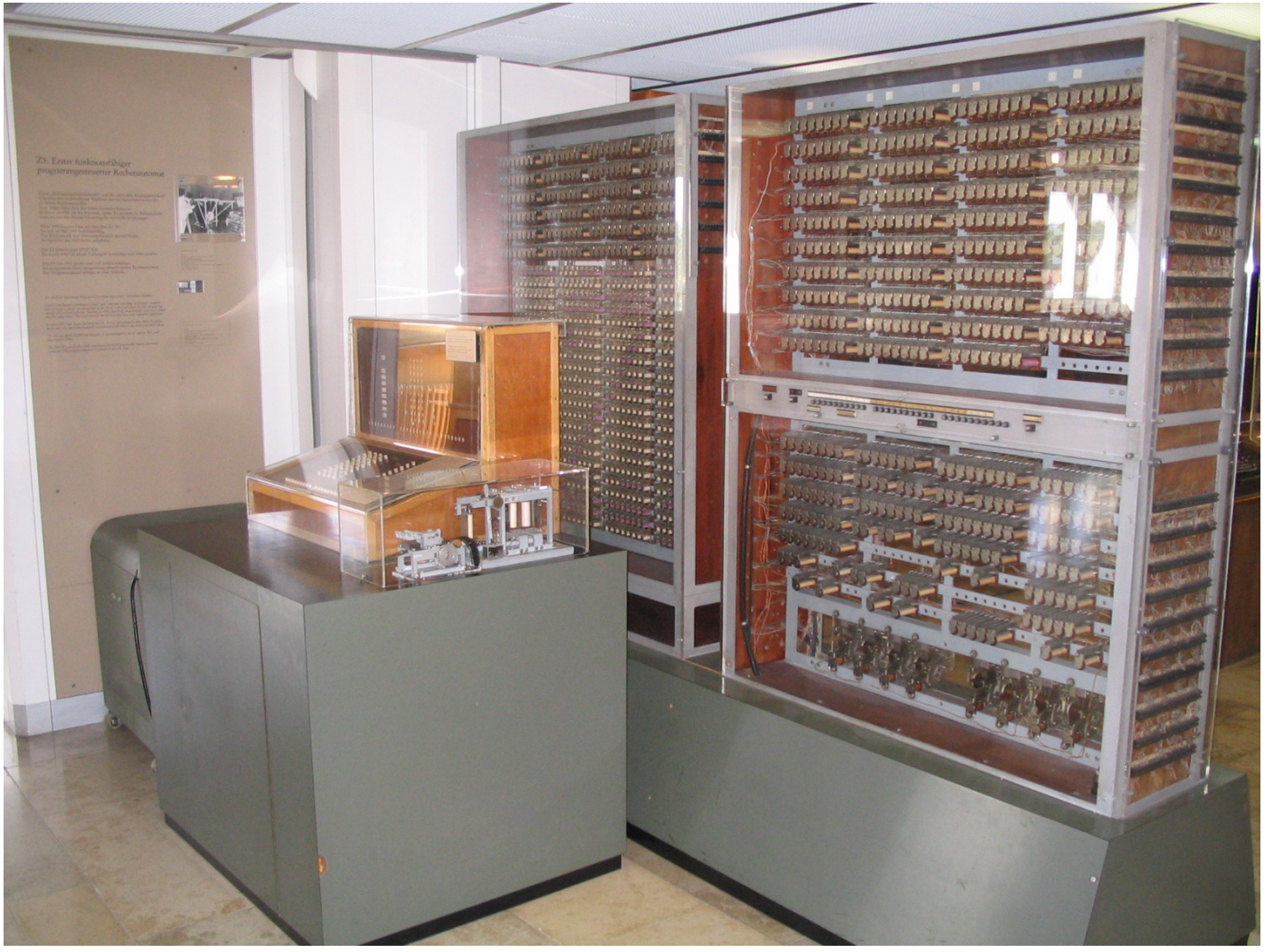
Week 9: LU Factorization: this is one of the most important operation behind solvers of linear systems and related scientific computing software

Week 10: Stencil computation: this parallel computing scheme is used in many algorithms for numerical analysis, like discritization of differential equations; we will focus on the discritization of the heat equation

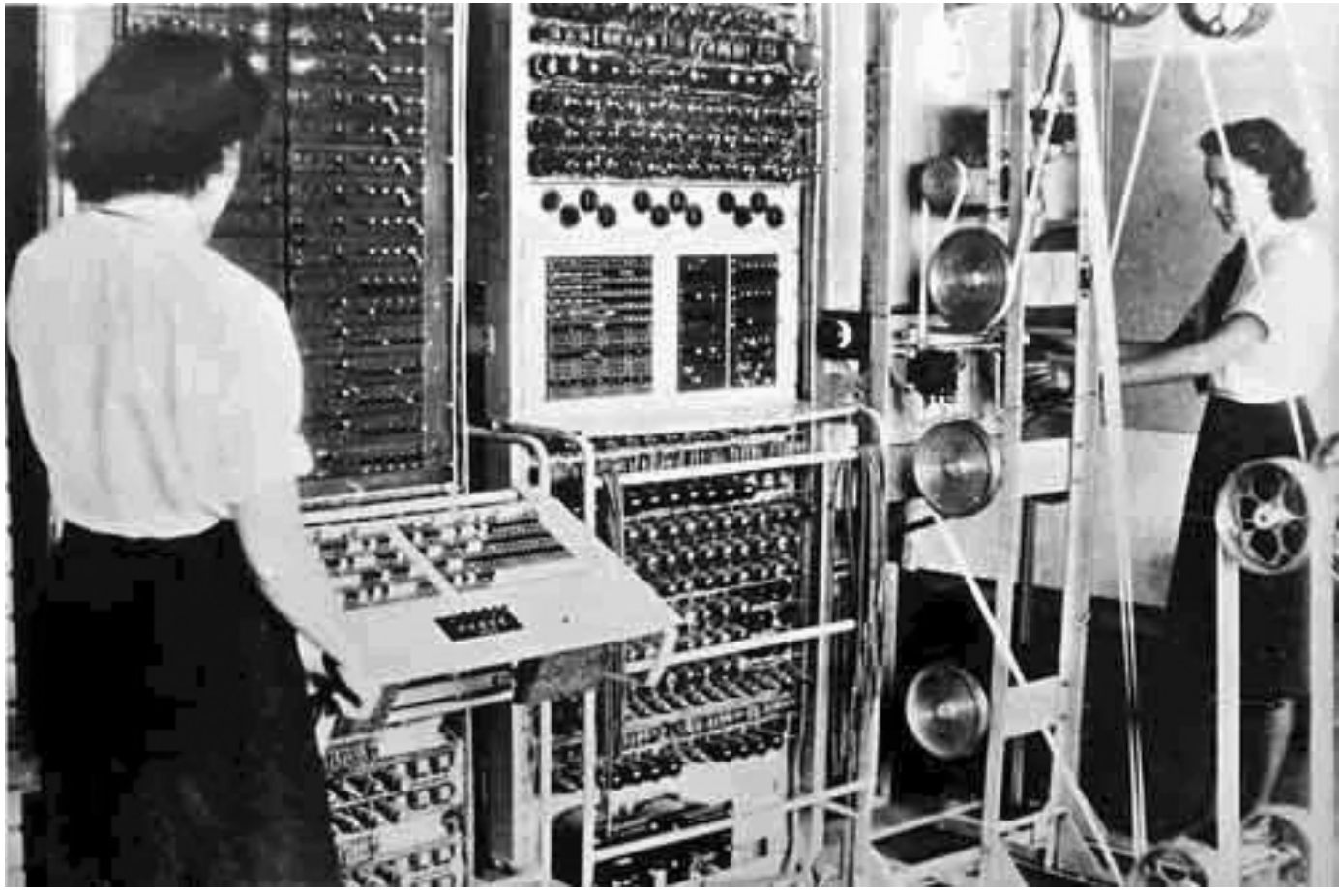
Week 11: Genetic and simulation algorithms: this other parallel computing scheme is used in applications dealing with vast amount of data; we will focus on Barnes-Hut Algorithm for N-Particle Interactions

Weeks 12: Parallel graph algorithms: designing efficient algorithms often imply to choose appropriate data-structures, which is highly non-trivial in the case of parallel computing; this topic will introduce important techniques for this goal

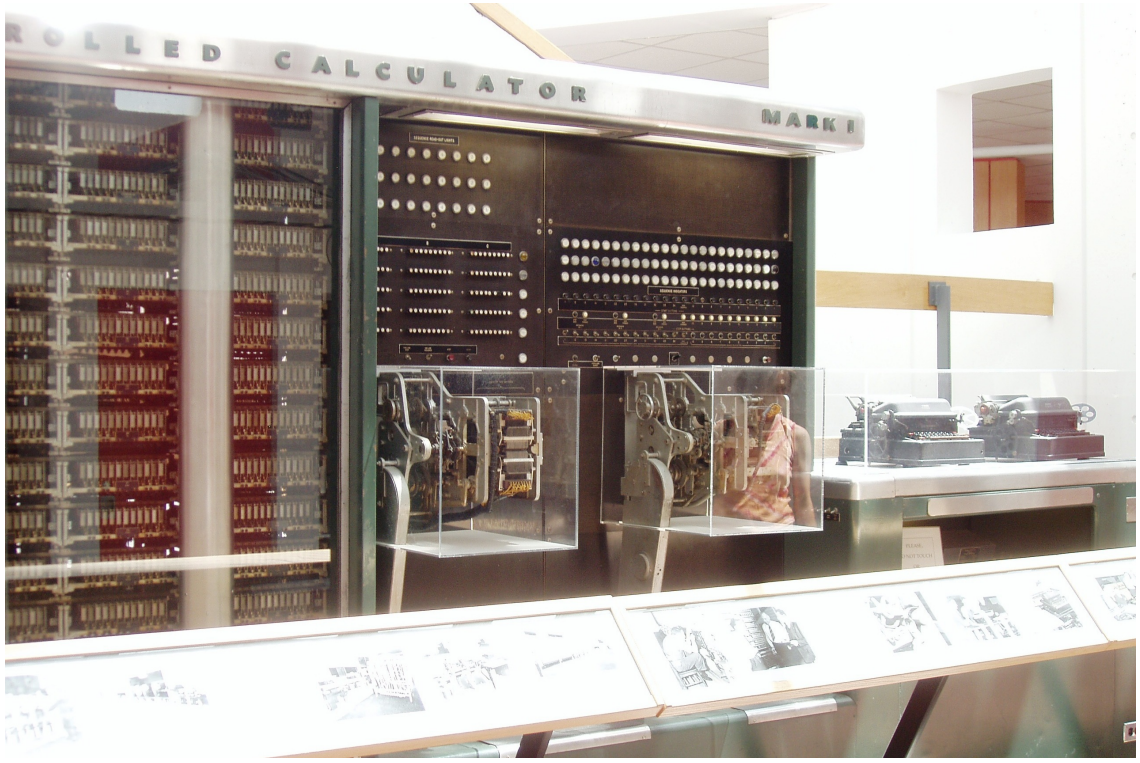
Plan



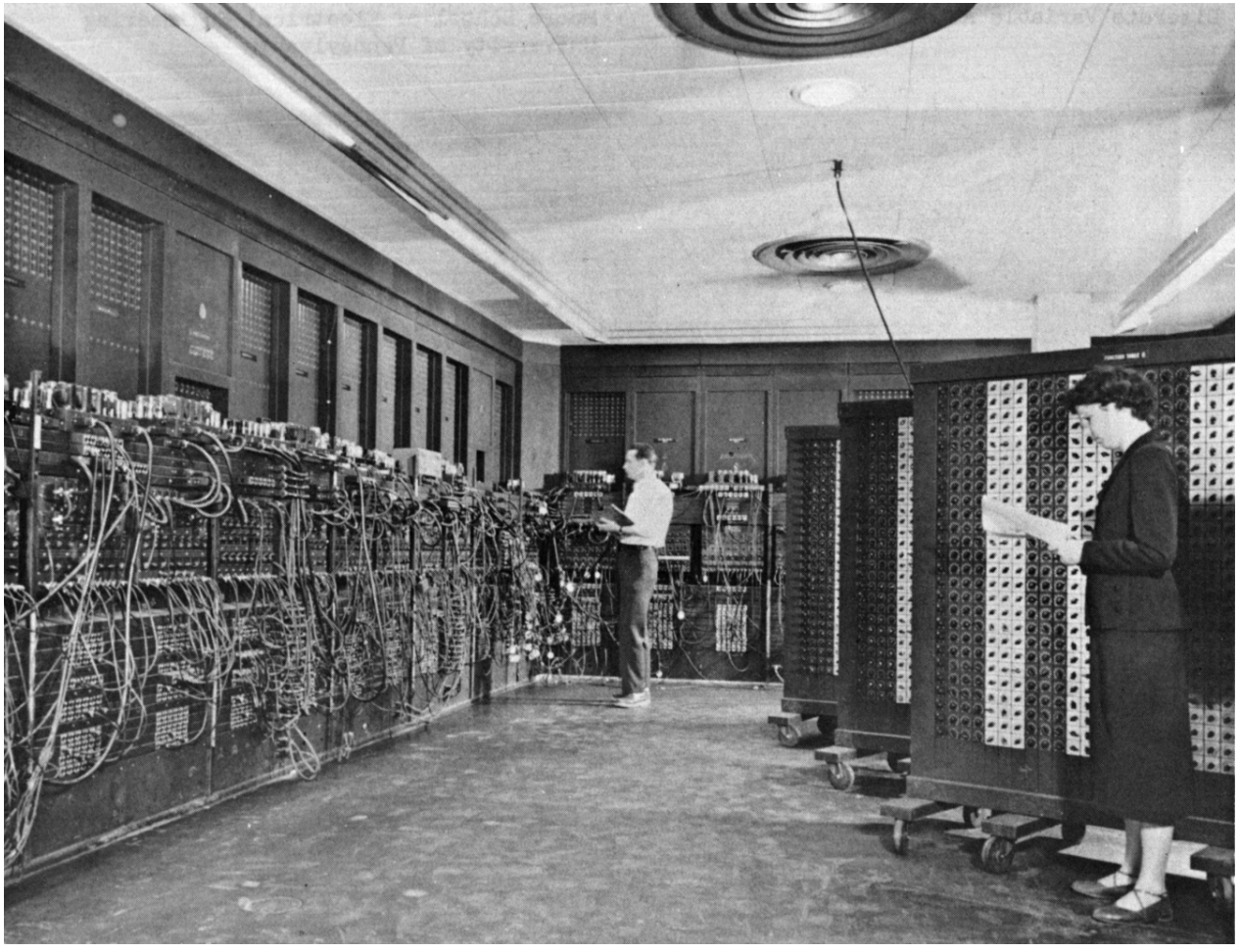
Konrad Zuse's Z3 electro-mechanical computer (1941, Germany). Turing complete, though conditional jumps were missing.



Colossus (UK, 1941) was the world's first totally electronic programmable computing device. But not Turing complete.



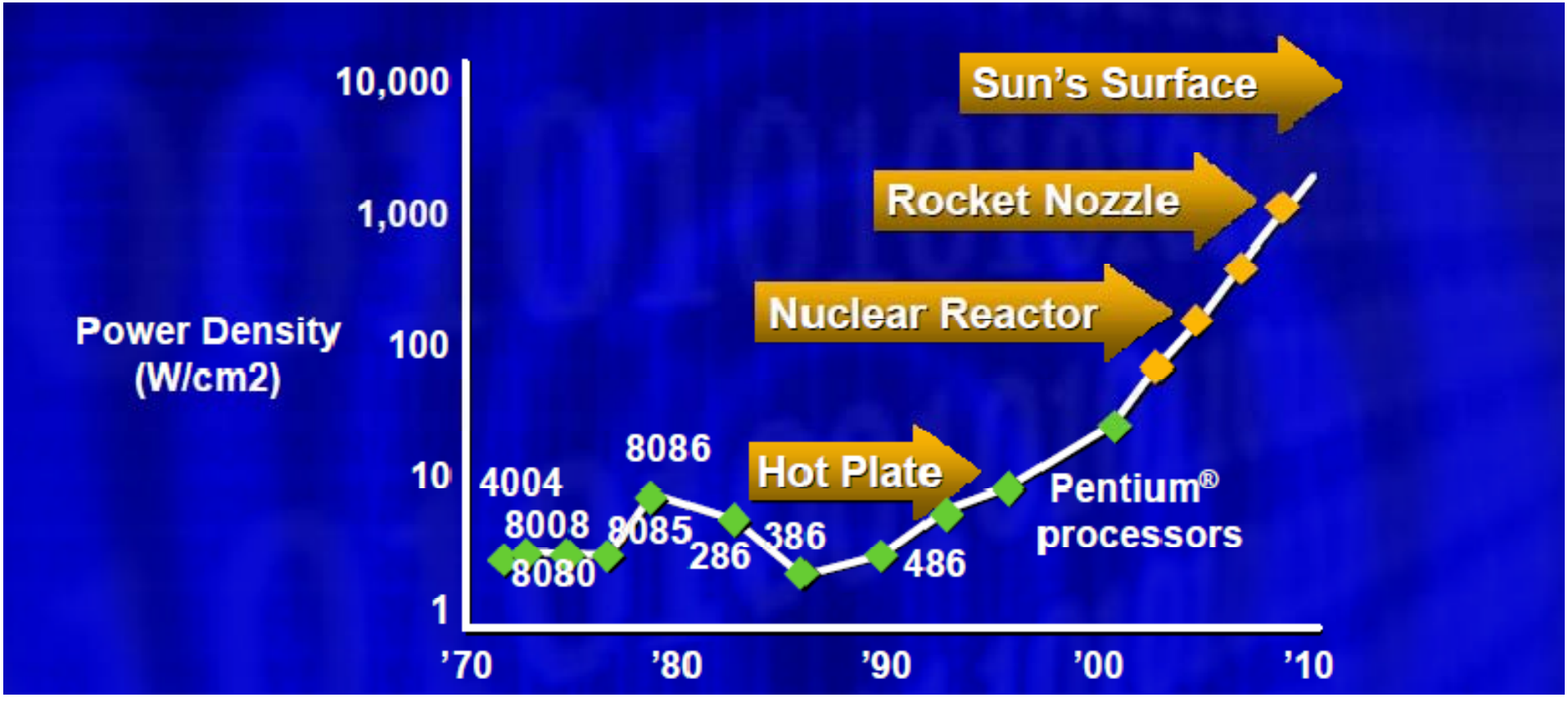
Harvard Mark I IBM ASCC (1944, US). Electro-mechanical computer (no conditional jumps and not Turing complete). It could store 72 numbers, each 23 decimal digits long. It could do three additions or subtractions in a second. A multiplication took six seconds, a division took 15.3 seconds, and a logarithm or a trigonometric function took over one minute. A loop was accomplished by joining the end of the paper tape containing the program back to the beginning of the tape (literally creating a loop).



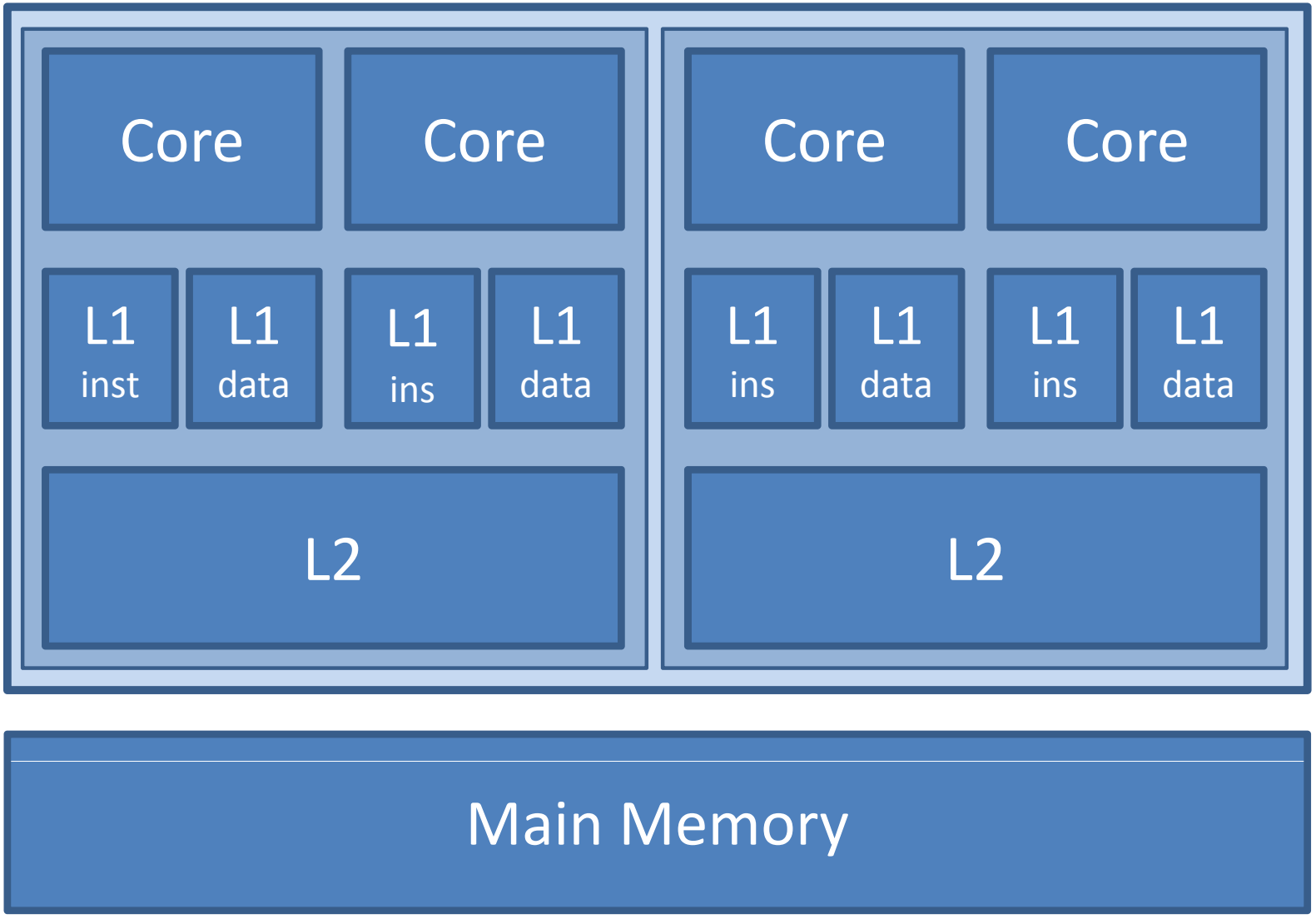
Electronic Numerical Integrator And Computer (ENIAC). The first general-purpose, electronic computer. It was a Turing-complete, digital computer capable of being reprogrammed and was running at 5,000 cycles per second for operations on the 10-digit numbers.

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

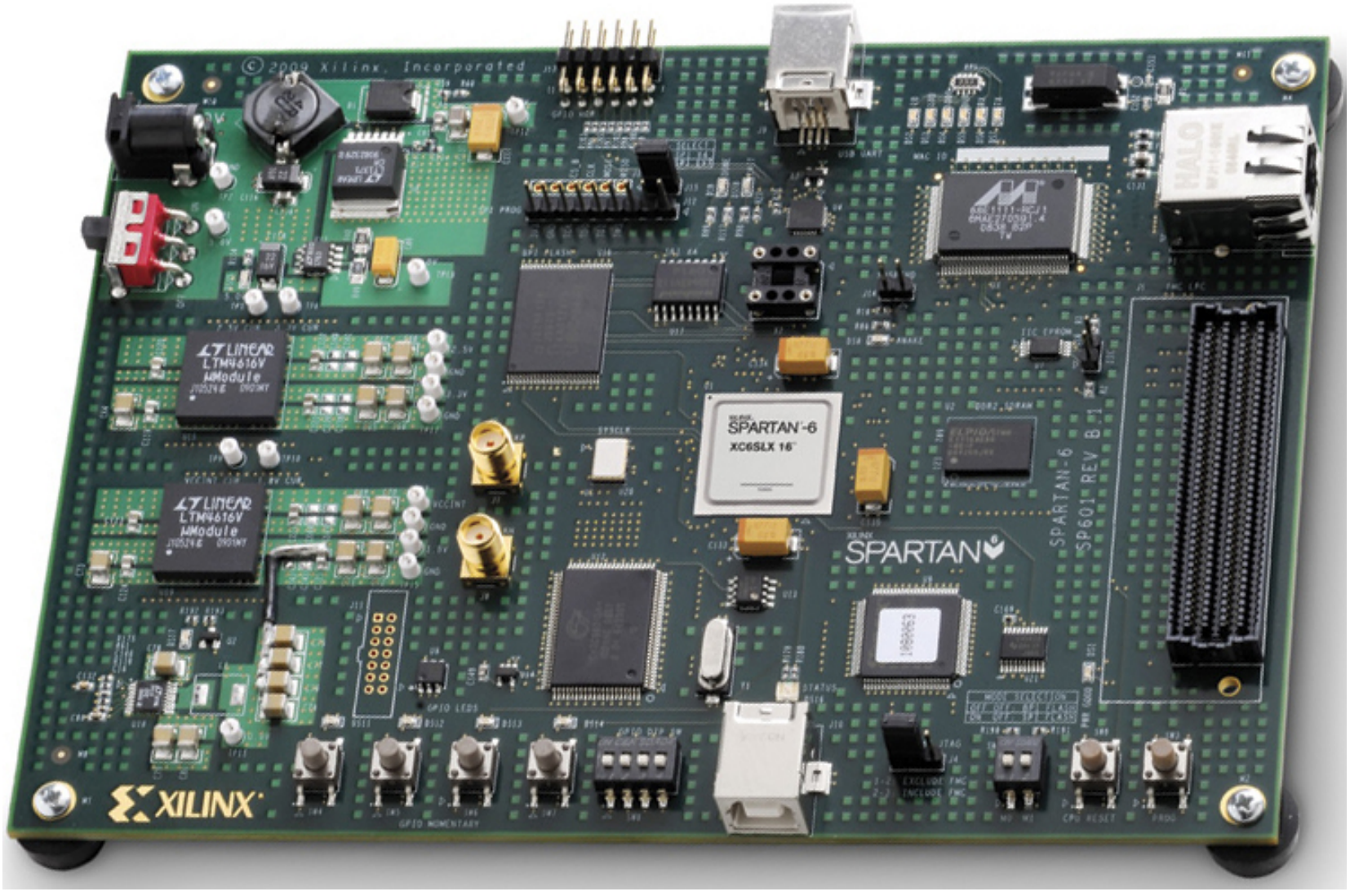
The Pentium Family.


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

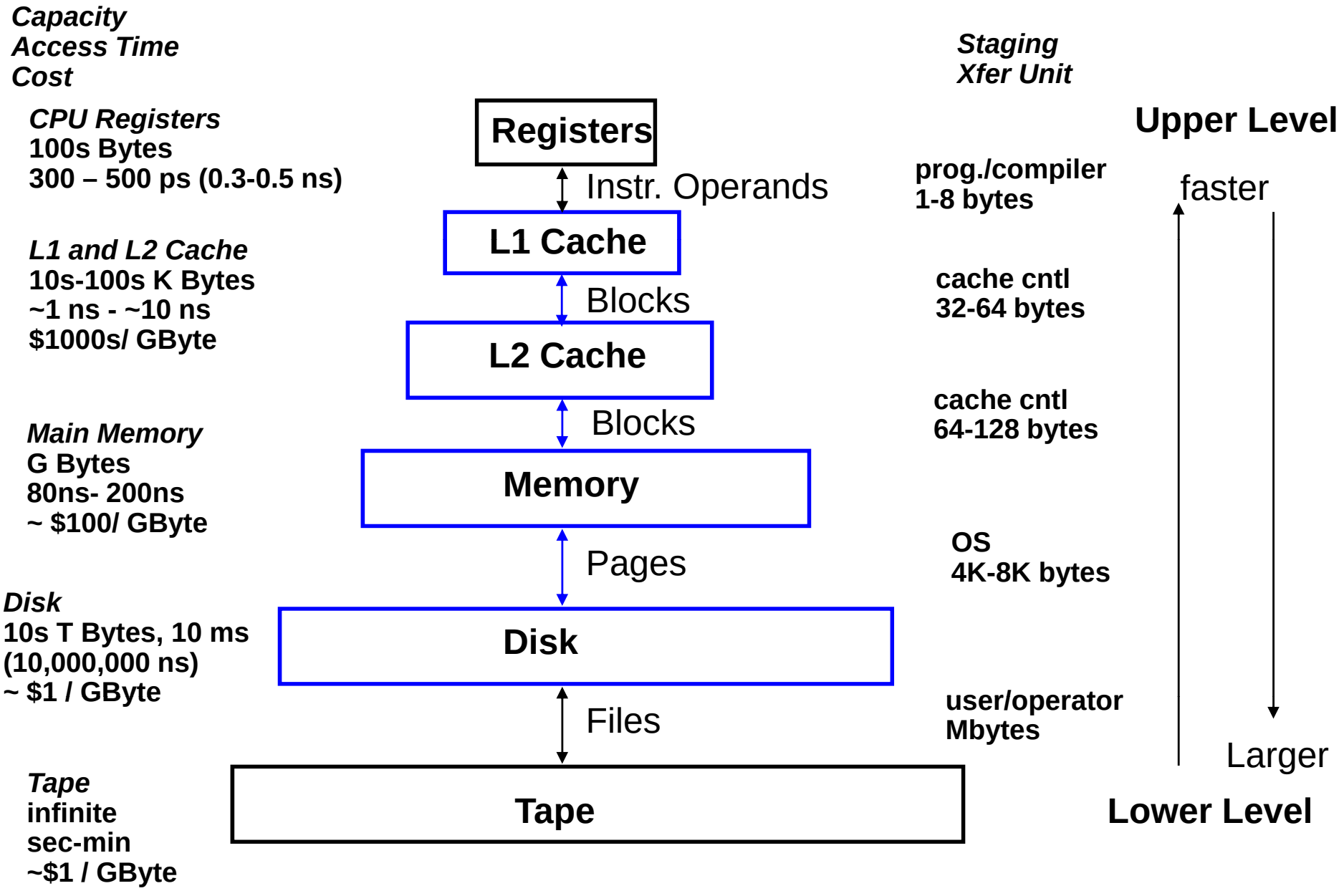

Core
Core
Core
Core
L1
inst
L1
data
L1
ins
L1
data
L1
ins
L1
data
L1
ins
L1
data
L2
L2
Main Memory


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


© 2009 Xilinx, Incorporated
LT LINEAR
LTM4616V
µModule
LT LINEAR
LTM4616V
µModule
SPARTAN-6
XC6SLX 16
SPARTAN
SP601 REV B1
XILINX

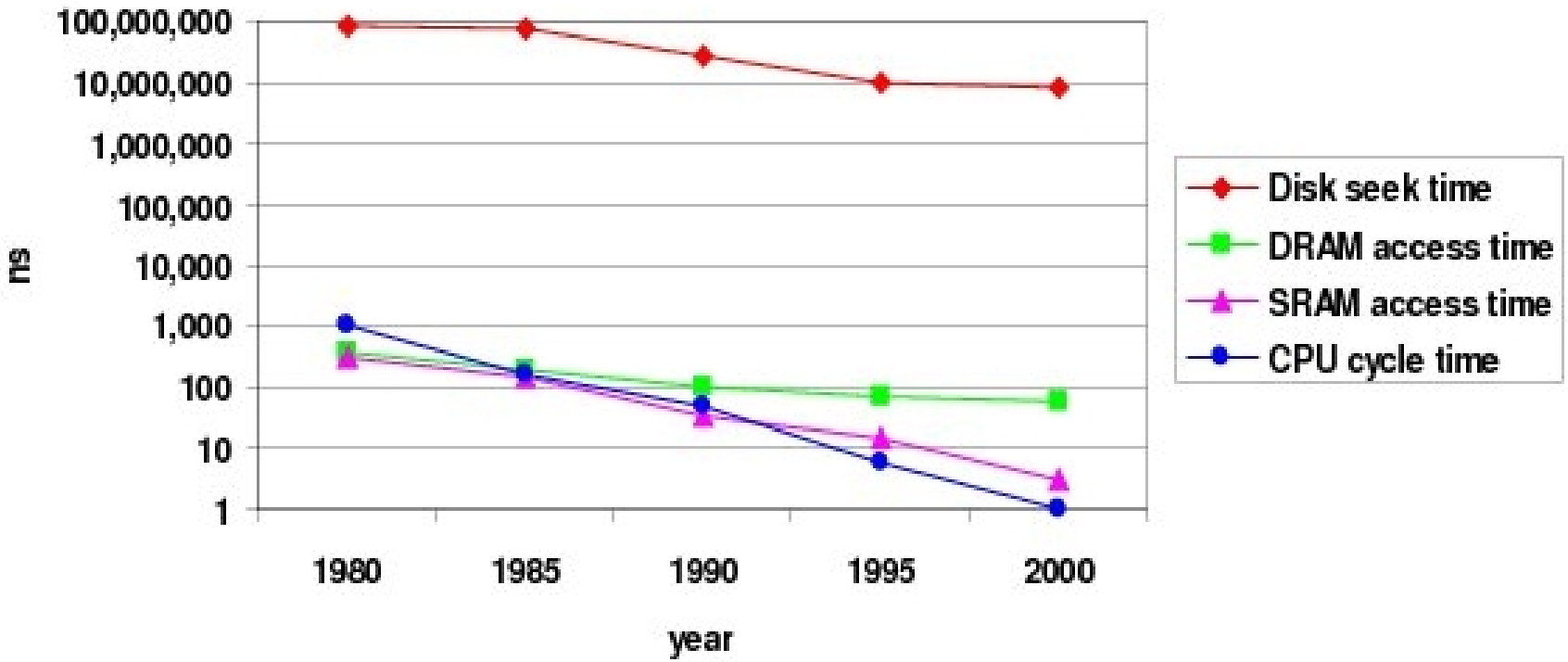
L1 Data Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L2 Cache			
Size	Line Size	Latency	Associativty
6 MB	64 bytes	14 cycles	24-way

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

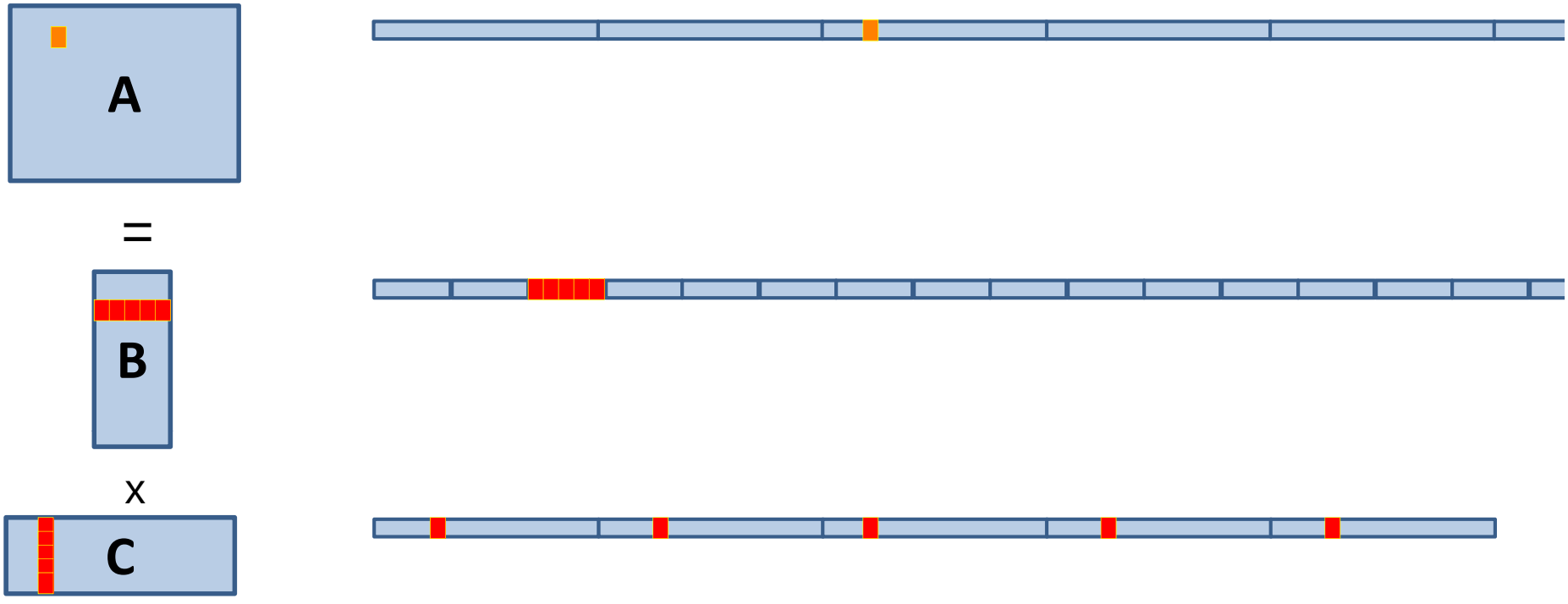
- Knowing that there is a performance problem: complexity estimates, performance analysis software tools, read the generated assembly code, scalability testing, comparisons to similar programs, experience and curiosity!
- Establishing the leading cause of the problem: examine the algorithm, the data 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

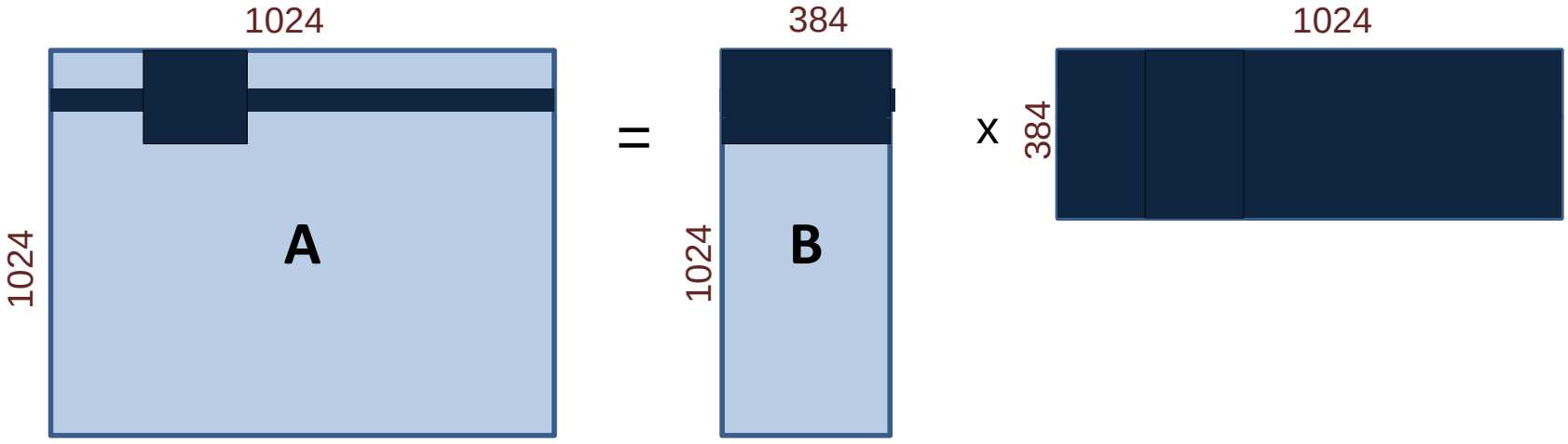


- Contiguous accesses are better:
  - Data fetch as cache line (Core 2 Duo 64 byte L2 Cache line)
  - With contiguous data, a single cache fetch supports 8 reads of doubles.
  - Transposing the matrix `C` should reduce L1 cache misses!

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.