CS4402-9535: Parallel and Distributed Systems

Marc Moreno Maza

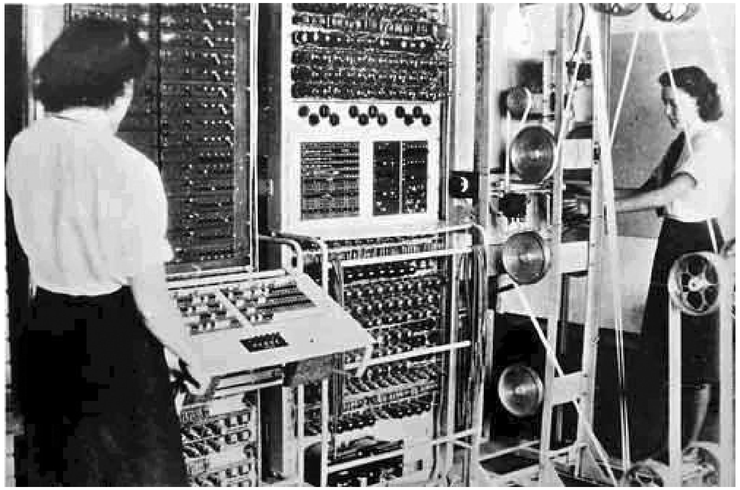
University of Western Ontario, London, Ontario (Canada)

CS4402-9535

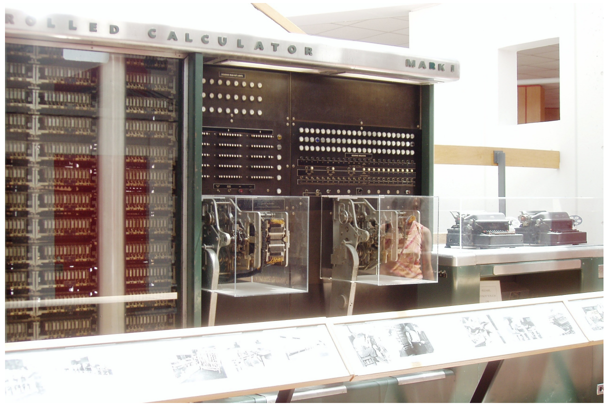
Plan

Plan

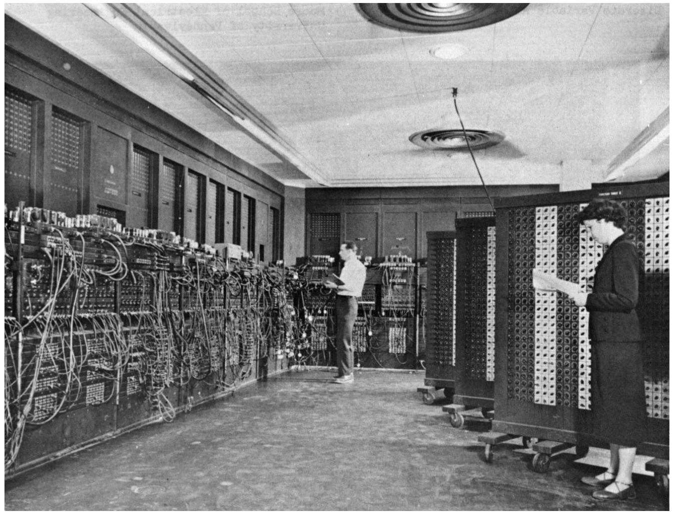
Konrad Zuse's Z3 electro-mechanical computer (1941, Germany). Turing complete, though conditional jumps were missing.



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).

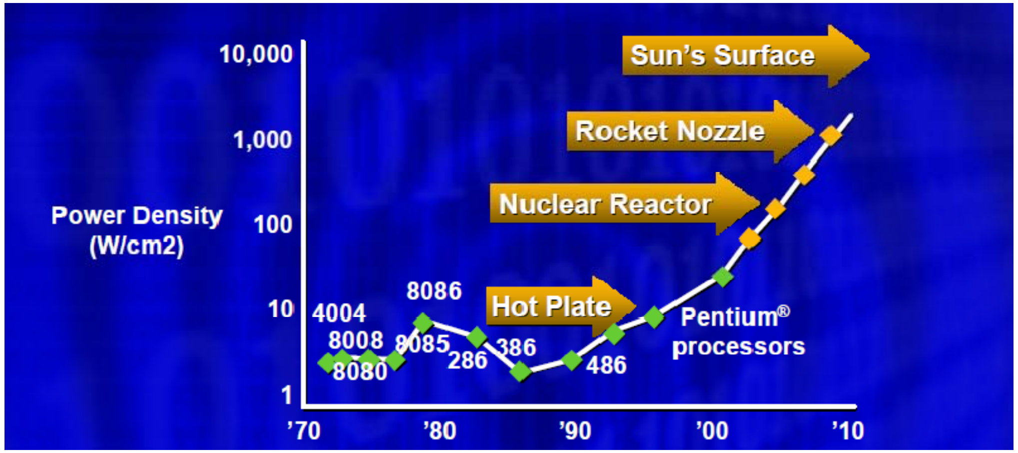


Electronic Numerical Integrator And Computer (ENIAC). The first general-purpose, electronic computer. It was a Turing-complete, digital computer capable of being reprogrammed and was running at 5,000 cycles per second for operations on the 10-digit numbers.

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

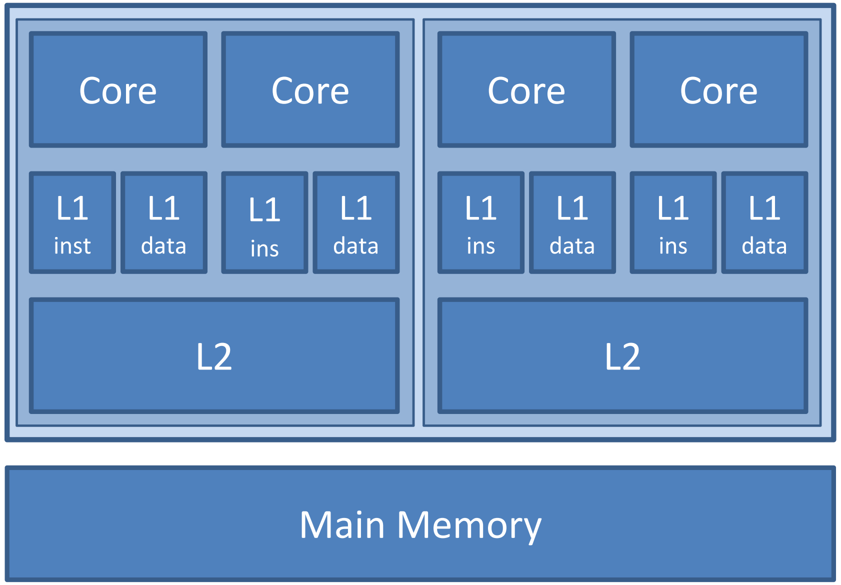
The Pentium Family.

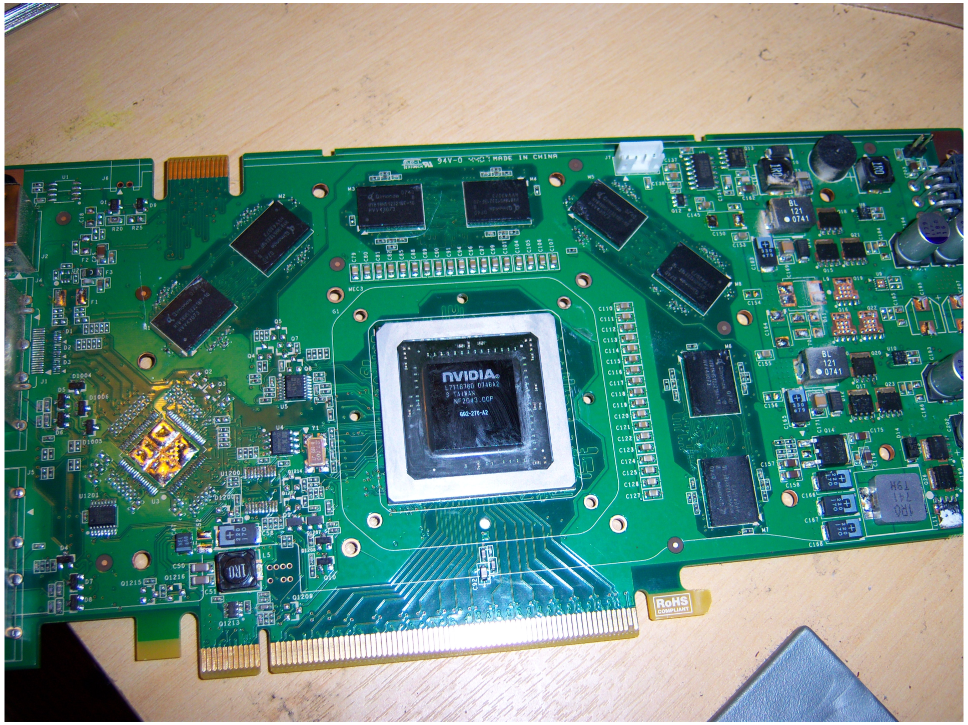
Hardware Acceleration Technologies

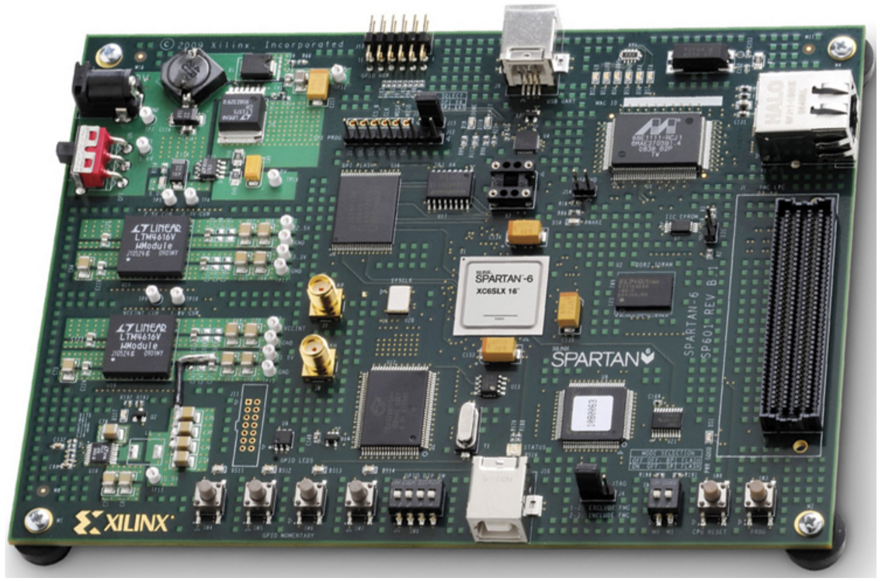


Hardware Acceleration Technologies

Hardware Acceleration Technologies

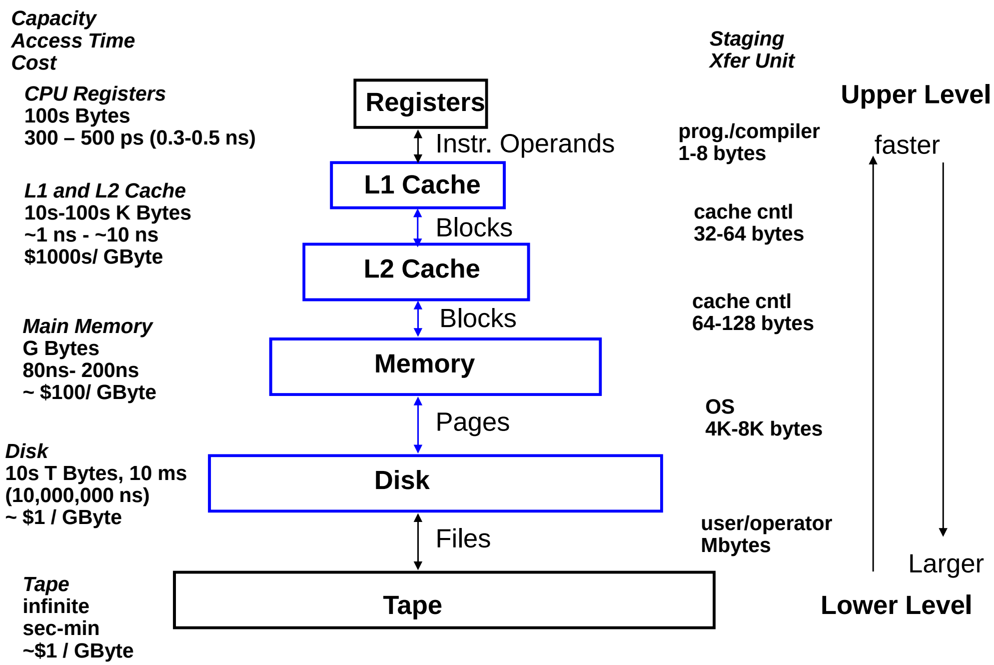






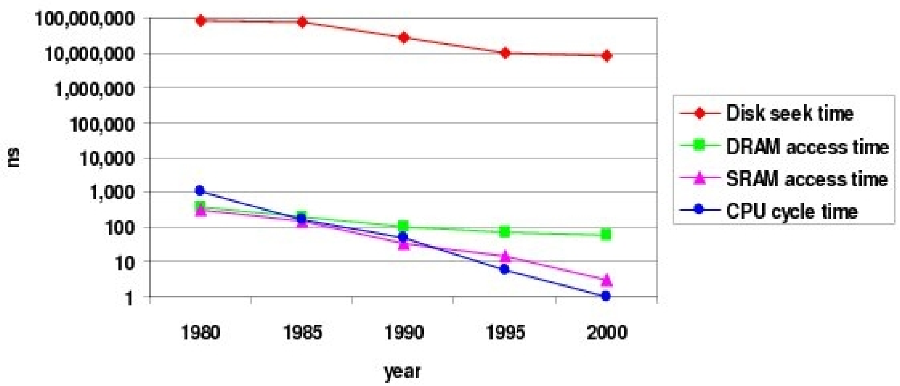
L1 Data Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L2 Cache			
Size	Line Size	Latency	Associativty
6 MB	64 bytes	14 cycles	24-way

Typical cache specifications of a multicore in 2008.



The CPU-Memory Gap

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



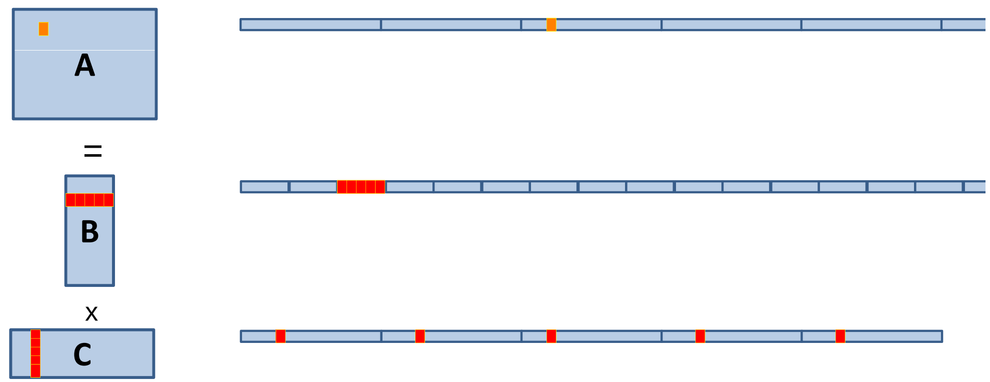
Once uopn a time, every thing was slow in a computer . . .

Plan

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;
        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

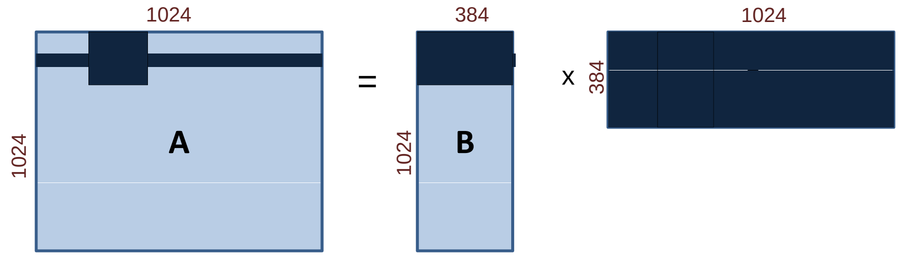


- Contiguous accesses are better:
  - Data fetch as cache line (Core 2 Duo 64 byte per cache line)
  - With contiguous data, a single cache fetch supports 8 reads of doubles.
  - Transposing the matrix C should reduce L1 cache misses!

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;
        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,y);
         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;
        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.

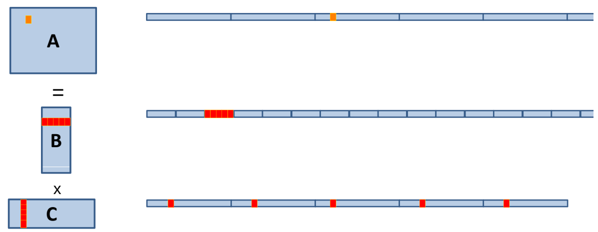
Other performance counters

Hardware count events

- CPI Clock cycles Per Instruction: the number of clock cycles that happen when an instruction is being executed. With pipelining we can improve the CPI by exploiting instruction level parallelism
- L1 and L2 Cache Miss Rate.
- Instructions Retired: In the event of a misprediction, instructions that were scheduled to execute along the mispredicted path must be canceled.

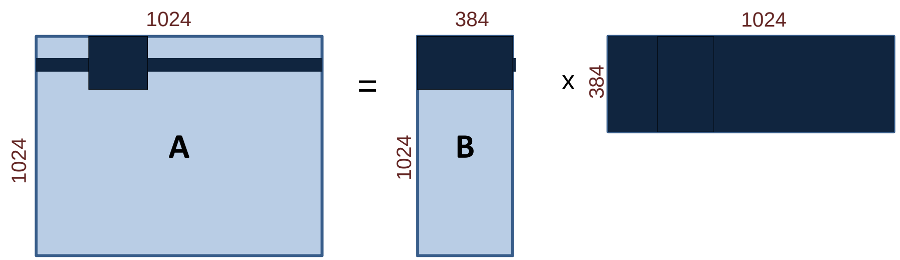
	CPI		L1 Miss Rate		L2 Miss Rate	Percent SSE Instructions	Instructions Retired	
In C	4.78		0.24		0.02	43%	13,137,280,000	
		5x		2x				1x
Transposed	1.13		0.15		0.02	50%	13,001,486,336	
		3x		8x				0.8x
Tiled	0.49		0.02		0	39%	18,044,811,264	

Analyzing cache misses in the naive and transposed multiplication



- Let A, B and C have format (m, n), (m, p) and (p, n) respectively.
- A is scanned once, so mn/L cache misses if L is the number of coefficients per cache line.
- B is scanned n times, so mnp/L cache misses if the cache cannot hold a row.
- C is accessed "nearly randomly" (for m large enough) leading to mnp cache misses.
- Since $2m\,n\,p$ arithmetic operations are performed, this means roughly **one cache miss per flop!**
- If C is transposed, then the ratio improves to 1 for L.

Analyzing cache misses in the tiled multiplication



- Let A, B and C have format (m, n), (m, p) and (p, n) respectively.
- Assume all tiles are square of order b and three fit in cache.
- If C is transposed, then loading three blocks in cache cost $3b^2/L$.
- This process happens n^3/b^3 times, leading to $3n^3/(bL)$ cache misses.
- Three blocks fit in cache for $3b^2 < Z$, if Z is the cache size.
- So $O(n^3/(\sqrt{Z}L))$ cache misses, if b is well chosen, which is optimal.

Plan

Cilk and CilkPlus

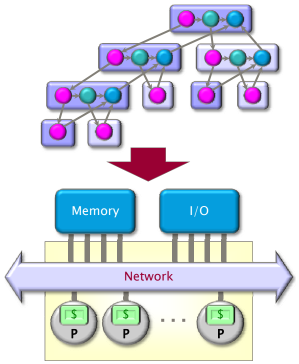
- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo.
- `Cilk` has been integrated into Intel C compiler under the name CilkPlus, see `http://www.cilk.com/`
- `CilkPlus` (resp. `Cilk`) is a small set of linguistic extensions to C++ (resp. C) supporting fork-join parallelism
- Both `Cilk` and `CilkPlus` feature a provably efficient work-stealing scheduler.
- `CilkPlus` provides a hyperobject library for parallelizing code with global variables and performing reduction for data aggregation.
- `CilkPlus` includes the `Cilkscreen` race detector and the `Cilkview` performance analyzer.

Nested Parallelism in CilkPlus

```
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
- CilkPlus keywords `cilk_spawn` and `cilk_sync` grant permissions for parallel execution. They do not command parallel execution.

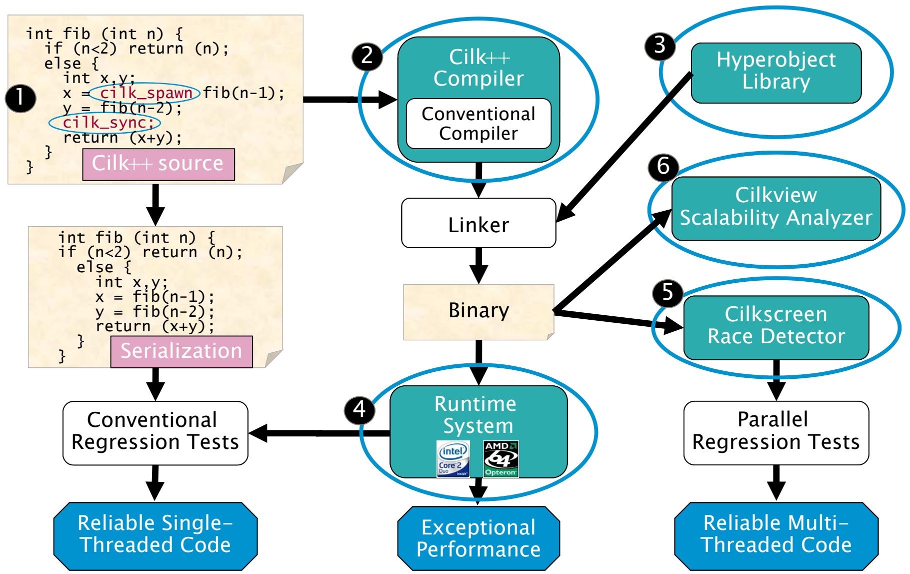
Scheduling



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 CilkPlus Platform



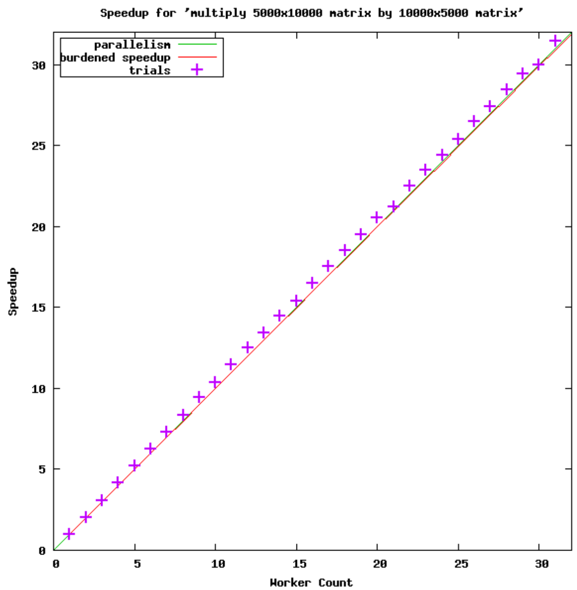
Benchmarks for the parallel version of the divide-n-conquer mm

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

Benchmarks using Cilkview



Plan

1. Hardware Acceleration Technologies
2. Optimizing Code for Data Locality: A Case Study
3. Multicore Programming
4. CS4402-9535 Course Outline

Course Topics

Week 1: Introduction to Multicore Programming

Week 2: Multithreaded Parallelism and the CilkPlus concurrency platform

Week 3: Analysis of Multithreaded Algorithms

Week 4: Issues with data locality and code parallelization

Week 5: Cache complexity

Week 6: Synchronizing without Locks and Concurrent Data Structures

Week 7: Pipelining

Weeks 8: CUDA Programming model

Week 9-10: CUDA Implementation on the GPU

Week 11: Code optimization with CUDA

Weeks 12: Multiprocessed parallelism, message passing (MPI)

Week 13: Course project presentations