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.
## L1 Data Cache

<table>
<thead>
<tr>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 KB</td>
<td>64 bytes</td>
<td>3 cycles</td>
<td>8-way</td>
</tr>
</tbody>
</table>

## L1 Instruction Cache

<table>
<thead>
<tr>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 KB</td>
<td>64 bytes</td>
<td>3 cycles</td>
<td>8-way</td>
</tr>
</tbody>
</table>

## L2 Cache

<table>
<thead>
<tr>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 MB</td>
<td>64 bytes</td>
<td>14 cycles</td>
<td>24-way</td>
</tr>
</tbody>
</table>

Hardware Acceleration Technologies

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

Software Performance Engineering

Plan

1. Hardware Acceleration Technologies
2. Software Performance Engineering
3. A Case Study: Matrix Multiplication
4. Multicore Programming
5. CS4402-9535 Course Outline

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!
A typical matrix multiplication C code

```c
#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!
A Case Study: Matrix Multiplication

### Transposing for optimizing spatial locality

```c
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**

- Computing a `32 \times 32`-block of `A`, so computing again 1024 coefficients: 1024 accesses in `A`, `384 \times 32` in `B` and `32 \times 384` in `C`. Total = 25,600.
- The iteration space is traversed so as to reduce memory accesses.

### Blocking for optimizing temporal locality

```c
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

```c
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;
}
```
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)

<table>
<thead>
<tr>
<th>$n$</th>
<th>naive</th>
<th>transposed</th>
<th>speedup</th>
<th>$64 \times 64$-tiled</th>
<th>speedup</th>
<th>t. &amp; t.</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>7</td>
<td>3</td>
<td></td>
<td>7</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>26</td>
<td>43</td>
<td>6.81</td>
<td>1928</td>
<td>0.936</td>
<td>187</td>
<td>9.65</td>
</tr>
<tr>
<td>512</td>
<td>1805</td>
<td>265</td>
<td>6.62</td>
<td>14020</td>
<td>1.76</td>
<td>1490</td>
<td>16.59</td>
</tr>
<tr>
<td>1024</td>
<td>24723</td>
<td>3730</td>
<td>9.11</td>
<td>112298</td>
<td>2.41</td>
<td>11960</td>
<td>22.69</td>
</tr>
<tr>
<td>2048</td>
<td>271446</td>
<td>29767</td>
<td>9.83</td>
<td>1009445</td>
<td>2.32</td>
<td>101264</td>
<td>23.15</td>
</tr>
<tr>
<td>4096</td>
<td>2344594</td>
<td>238453</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Timings are in milliseconds.

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

### 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.

<table>
<thead>
<tr>
<th>#core</th>
<th>Elision (s)</th>
<th>Parallel (s)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>420.906</td>
<td>51.365</td>
<td>8.19</td>
</tr>
<tr>
<td>16</td>
<td>432.419</td>
<td>25.845</td>
<td>16.73</td>
</tr>
<tr>
<td>24</td>
<td>413.681</td>
<td>17.361</td>
<td>23.83</td>
</tr>
<tr>
<td>32</td>
<td>389.300</td>
<td>13.051</td>
<td>29.83</td>
</tr>
<tr>
<td>Plan</td>
<td>Course Topics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware Acceleration Tech.</td>
<td><strong>Week 1:</strong> Introduction to Multicore Programming</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Week 2:</strong> Multithreaded Parallelism and the CilkPlus concurrency platform</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Week 3:</strong> Analysis of Multithreaded Algorithms</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Week 4:</strong> Issues with data locality and code parallelization</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Week 5:</strong> Cache complexity</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Week 6:</strong> Synchronizing without Locks and Concurrent Data Structures</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Week 7:</strong> Parallelism overheads</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Weeks 8:</strong> CUDA Programming model</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Week 9-10:</strong> CUDA Implementation on the GPU</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Week 11:</strong> Code optimization with CUDA</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Weeks 12:</strong> Multiprocessed parallelism, message passing (MPI)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Week 13:</strong> Course project presentations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software Performance Eng.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A Case Study: Matrix Multip.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multicore Programming</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS402-9535 Course Outline</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>