CS4402-9535: Parallel and Distributed Systems

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS4402-9535
Plan

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

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

The Pentium Family.
Hardware Acceleration Technologies
Hardware Acceleration Technologies

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  

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  

closer  

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  

Lower Level  

Larger  

Registers  

Instr. Operands  

L1 Cache  

Blocks  

L2 Cache  

Memory  

Pages  

Disk  

Files  

Tape  

User/Operator  

Mbytes  

Larger  

Upper Level  

closer  

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  

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  

Lower Level  

Larger
Once upon a time, everything was slow in a computer...
Plan

1. Hardware Acceleration Technologies

2. Optimizing Code for Data Locality: A Case Study

3. Multicore Programming

4. CS4402-9535 Course Outline
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; *B; *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,y);
    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

```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.
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;
}
```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(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 += 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(Cx,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 (Quad-core Intel i7-3630QM CPU @ 2.40GHz L2 cache 6144 KB, 8 GBytes of RAM)

<table>
<thead>
<tr>
<th>$n$</th>
<th>naive</th>
<th>transposed</th>
<th>$8 \times 8$-tiled</th>
<th>t. &amp; t.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>7854</td>
<td>1086</td>
<td>1105</td>
<td>999</td>
</tr>
<tr>
<td>2048</td>
<td>8335</td>
<td>8646</td>
<td>10166</td>
<td>7990</td>
</tr>
<tr>
<td>4096</td>
<td>747100</td>
<td>69149</td>
<td>100538</td>
<td>69745</td>
</tr>
<tr>
<td>8192</td>
<td>6914349</td>
<td>546585</td>
<td>823525</td>
<td>562433</td>
</tr>
</tbody>
</table>

Timings are in milliseconds.

The cache-oblivious multiplication (more on this later) and the titled multiplication have similar performance.
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.

<table>
<thead>
<tr>
<th></th>
<th>CPI</th>
<th>L1 Miss Rate</th>
<th>L2 Miss Rate</th>
<th>Percent SSE</th>
<th>Instructions Retired</th>
</tr>
</thead>
<tbody>
<tr>
<td>In C</td>
<td>4.78</td>
<td>0.24</td>
<td>0.02</td>
<td>43%</td>
<td>13,137,280,000</td>
</tr>
<tr>
<td>Transposed</td>
<td>1.13</td>
<td>0.15</td>
<td>0.02</td>
<td>50%</td>
<td>13,001,486,336</td>
</tr>
<tr>
<td>Tiled</td>
<td>0.49</td>
<td>0.02</td>
<td>0</td>
<td>39%</td>
<td>18,044,811,264</td>
</tr>
</tbody>
</table>
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 $2mnp$ arithmetic operations are performed, this means roughly one cache miss per flop!
- If $C$ is transposed, then the ratio improves to 1 for $L$. 

$A = B \times C$
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

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

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

1. Cilk++ source
2. Cilk++ Compiler
3. Hyperobject Library
4. Conventional Regression Tests
5. Cilkview Scalability Analyzer
6. Cilkscreen Race Detector
7. Linker
8. Binary
9. Cilkview Scalability Analyzer
10. Cilkscreen Race Detector
11. Conventional Compiler
12. Exceptional Performance
13. Reliable Single-Threaded Code
14. Reliable Multi-Threaded Code
15. Parallel Regression Tests
### Benchmarks for the parallel version of the divide-n-conquer \textbf{mm}

Multiplying a $4000 \times 8000$ matrix by a $8000 \times 4000$ 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>
</tbody>
</table>
Benchmarks using Cilkview

Speedup for 'multiply 5000x10000 matrix by 10000x5000 matrix'

- parallelism
- burdened speedup
- trials
Plan

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

- Some familiarity with algorithms and their analysis.
- Elementary linear algebra (matrix multiplication).
- Ideas about multithreaded programming.
- Some ideas about multi-core processors and GPUs.
What are the objectives of this course?

1. Understand why data locality can have a huge impact on code performances.
2. Acquire techniques for analyzing and improving data locality.
3. Understand the concepts of work, span, parallelism, burdened parallelism in multithreaded programming.
4. Acquire techniques for analyzing and improving parallelism in multithreaded programming.
5. Understand issues related to parallelism overheads in GPU programming.
6. Acquire techniques for reducing parallelism overheads of a GPU kernel.
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 (Cilk-P, TBB)
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
High-performance computing and symbolic computation

www.bpaslib.org

www.metafork.org

www.cumodp.org

www.regularchains.org
Acknowledgments and references

Acknowledgments.
- Charles E. Leiserson (MIT), Matteo Frigo (Axis Semiconductor) Saman P. Amarasinghe (MIT) and Cyril Zeller (NVIDIA) for sharing with me the sources of their course notes and other documents.
- My past and current graduate students, in particular: Changbo Chen (Chinese Academy of Science) Xiaohui Chen (UWO), Svyatoslav Covano (UWO & École Polytechnique) Anisul Sardar Haque (Mississauga), Xin Li (U. Carlos III), Farnam Mansouri (Microsoft), Wei Pan (Intel Corp.) and Ning Xie (UWO) for their contribution to the materials presented in this tutorial.

References.
- http://www.csd.uwo.ca/~moreno/HPC-Resources.html