Optimizing Algorithms and Code for Data Locality and Parallelism

Marc Moreno Maza

University of Western Ontario, Canada

Chengdu HPC Summer School
July 20-24, 2015
Canada in 4 pictures

[Map of Canada]

[Space station module]

[Ice hockey players celebrating]

[Wooden sculpture]
High-performance computing and symbolic computation

Research themes

- **Symbolic computation:** computing exact solutions of algebraic problems on computers with applications to mathematical sciences and engineering.

- **High-performance computing:** making best use of modern computer architectures, in particular hardware accelerators (multi-core processors, graphics processing units).

Research projects

- The *RegularChains* library: solving systems of algebraic equations and integrated into the computer algebra system *Maple*.

- The *Basic Polynomial Algebra Subroutines* (BPAS) and *CUDA Modular Polynomial* (CUMODP) libraries: hardware accelerator support for symbolic computation.

- The *Meta_Fork* compilation framework: a programming environment for hardware accelerators, supported by IBM Toronto Labs.
Current students and alumni

**Current students**
Parisa Alvandi, Ning Xie, Xiaohui Chen, Li Zhang, Haowei Chen, Yiming Guan, Davood Mohajerani, Steven Thornton and Robert Moir.

**Alumni**
Moshin Ali (ANU, Australia) Jinlong Cai (Microsoft), Changbo Chen (CIGIT), Sardar Anisula Haque (CiTi) Zunaid Haque (IBM) François Lemaire (U. Lille 1, France) Xin Li (U. Carlos III, Spain) Wei Pan (Intel) Paul Vrbik (U. Newcastle, Australia) Yuzhen Xie (COT) . . .
Solving polynomial systems symbolically

\[ R := \text{PolynomialRing}([x, y, z]); F := [5x^2 + 2xz^2 + 5y^6 + 15y^4 + 5z^2 - 15y^5 - 5y^3]; \]

\[ \text{polynomial_ring} \]
\[ [5x^2 + 2xz^2 + 5y^6 + 15y^4 + 5z^2 - 15y^5 - 5y^3] \]

\[ \text{RealTriangularize}(F, R, \text{output} = \text{record}); \]
\[ \begin{cases} 5x^2 + 2z^2x + 5y^6 + 15y^4 - 5y^3 - 15y^5 + 5z^2 = 0 \\ 25y^6 - 75y^5 + 75y^4 - z^2 - 25y^3 + 25z^2 < 0 \\ \end{cases} \]

\[ \begin{cases} 5x + z^2 = 0 \\ 25y^6 - 75y^5 + 75y^4 - z^2 - 25y^3 + 25z^2 = 0 \\ 64z^4 - 1600z^2 + 25 > 0 \\ z \neq 0 \\ z - 5 \neq 0 \\ z + 5 \neq 0 \end{cases} \]

\[ \begin{cases} x = 0 \\ y - 1 = 0 \\ z = 0 \end{cases}, \quad \begin{cases} x = 0 \\ y = 0 \\ z = 0 \end{cases}, \quad \begin{cases} x + 5 = 0 \\ y - 1 = 0 \\ z - 5 = 0 \end{cases}, \quad \begin{cases} x + 5 = 0 \\ y = 0 \\ z + 5 = 0 \end{cases}, \quad \begin{cases} x + 5 = 0 \\ 5x + z^2 = 0 \\ 64z^4 - 1600z^2 + 25 = 0 \end{cases} \]

Figure: The \textit{RegularChains} solver designed in our UWO lab can compute the real solutions of any polynomial system exactly.
Our polynomial system solver is at the core of **Maple**

**Figure:** Maplesoft, the company developing **Maple**, demonstrates the *RegularChains* solver designed in our UWO lab, in order to advertise **Maple**.
Let $K$ be the maximum number of thread blocks along an anti-chain of the thread-block DAG representing the program $P$. Then the running time $T_P$ of the program $P$ satisfies:

$$T_P \leq \left( \frac{N(P)}{K} + L(P) \right) C(P),$$

where $C(P)$ is the maximum running time of local operations by a thread among all the thread-blocks, $N(P)$ is the number of thread-blocks and $L(P)$ is the span of $P$.

Our UWO lab develops mathematical models to make efficient use of hardware acceleration technology, such as GPUs and multi-core processors.
Our lab develops a compilation platform for translating parallel programs from one language to another; above we translate from OpenMP to CilkPlus through MetaFork.
High-performance computing: automatic parallelization

Serial dense univariate polynomial multiplication

```c
for(i=0; i<=n; i++){
    for(j=0; j<=n; j++)
        c[i+j] += a[i] * b[j];
}
```

Dependence analysis suggests to set \( t(i, j) = n - j \) and \( p(i, j) = i + j \). Then, the work is decomposed into blocks having good data locality.

GPU-like multi-threaded dense univariate polynomial multiplication

```c
meta_for (b=0; b<= 2 n / B; b++) {
    for (u=0; u<=min(B-1, 2*n - B * b); u++) {
        p = b * B + u;
        for (t=max(0,n-p); t<=min(n,2*n-p) ;t++)
            c[p] = c[p] + a[t+p-n] * b[n-t];
    }
}
```

We use symbolic computation to automatically translate serial programs to GPU-like programs.
Research projects with publicly available software

- Basic Polynomial Algebra Subprograms
  - www.bpaslib.org

- MetaFork
  - www.metafork.org

- CUMODP
  - www.cumodp.org

- Regular Chains
  - www.regularchains.org
Application to mathematical sciences and engineering

New Maplesoft project for Toyota leverages symbolic computation in control systems engineering

Maplesoft

February 4, 2013

Follows development of highly successful model simplification tools using Maplesoft technology

Waterloo, Canada; 4 February 2013: Maplesoft today announced that its ongoing partnership with Toyota Motor Engineering & Manufacturing North America, Inc. has expanded to include a new project. This project will see the leading car manufacturer use new symbolic computation methods in robust control design, with a strong focus on methods for linear, nonlinear, and parametric systems. Maplesoft technology is rooted in strong symbolic computation techniques, making it an appropriate choice for Toyota.

The main goal of the new Symbolic Control project is ground-breaking research that leads to the implementation of symbolic design and analysis methods for linear, nonlinear, and parametric robust control. The new research will allow developers to consider system nonlinearities, modeling inaccuracies, and parametric uncertainties in the design process. As a result, Toyota expects to shorten development time while maintaining high quality results.

This new project follows the completion of another successful Maplesoft project for Toyota, which saw high-end research and implementation of new methods for model simplification and preprocessing of high level acausal dynamical models. Simplification enables the conversion of high-level descriptive models into smaller executable models for faster execution and provides for better analysis, higher efficiency, and more accurate simulation. Model simplification allows engineers to focus on describing the physical properties of the system in an equation-based manner. It also makes use of the power of mathematical equations to better manage models, so engineers obtain the optimal results faster.

Figure: Toyota engineers use our software to design control systems
What is this mini-course about?

Optimizing algorithms and code

- Improving code performance is hard and complex.
- Requires a good understanding of the underlying algorithm and implementation environment (hardware, OS, compiler, etc.).
What is this mini-course about?

**Optimizing algorithms and code**
- Improving code performance is hard and complex.
- Requires a good understanding of the underlying algorithm and implementation environment (hardware, OS, compiler, etc.).

**Optimizing for data locality**
- Computer cache memories have led to introduce a new complexity measure for algorithms and new performance counters for code.
- Optimizing for data locality brings large speedup factors.
What is this mini-course about?

**Optimizing algorithms and code**
- Improving code performance is hard and complex.
- Requires a good understanding of the underlying algorithm and implementation environment (hardware, OS, compiler, etc.).

**Optimizing for data locality**
- Computer cache memories have led to introduce a new complexity measure for algorithms and new performance counters for code.
- Optimizing for data locality brings large speedup factors.

**Optimizing for parallelism**
- All recent home and office desktops/laptops are parallel machines; moreover “GPU cards bring supercomputing to the masses.”
- Optimizing for parallelism improves the use of computing resources.
- And optimizing for data locality is often a first step!
Once upon a time, everything was slow in a computer.
The second space race . . .
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 mini-course?

1. Understand why data locality can have a huge impact on code performances.
2. Acquire some ideas on how data locality can be analyzed and improved.
3. Understand the concepts of work, span, parallelism, burdened parallelism in multithreaded programming.
4. Acquire some ideas on how parallelism can be analyzed and improved in multithreaded programming.
5. Understand issues related to parallelism overheads in GPU programming.
6. Acquire some ideas on how to reduce parallelism overheads of a GPU kernel.
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: Yuzhen Xie (Critical Outcome Technologies Inc.) 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 mini-course.

References.

- The Implementation of the Cilk-5 Multithreaded Language by Matteo Frigo Charles E. Leiserson Keith H. Randall.
- The Cache Complexity of Multithreaded Cache Oblivious Algorithms by Matteo Frigo and Volker Strumpen.
- http://www.csd.uwo.ca/~moreno/HPC-Resources.html
Plan

1. Data locality and cache misses
   - Hierarchical memories and their impact on our programs
   - Cache complexity and cache-oblivious algorithms put into practice
   - A detailed case study: counting sort

2. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join multithreaded programming model
   - Anticipating parallelization overheads
   - Practical issues and optimization tricks

3. GPU programming
   - The CUDA programming and memory models

4. CUDA Programming: more details and examples

5. CUDA Programming: more details and examples
   - Tiled matrix multiplication in CUDA
   - Optimizing Matrix Transpose with CUDA
   - CUDA programming practices
Plan

1. Data locality and cache misses
   - Hierarchical memories and their impact on our programs
   - Cache complexity and cache-oblivious algorithms put into practice
   - A detailed case study: counting sort

2. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join multithreaded programming model
   - Anticipating parallelization overheads
   - Practical issues and optimization tricks

3. GPU programming
   - The CUDA programming and memory models

4. CUDA Programming: more details and examples

5. CUDA Programming: more details and examples
   - Tiled matrix multiplication in CUDA
   - Optimizing Matrix Transpose with CUDA
   - CUDA programming practices
Data locality and cache misses

Hierarchical memories and their impact on our programs

### Capacity
- **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

### Access Time
- **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

### 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
A **CPU cache** is an auxiliary memory which is smaller, faster memory than the main memory and which stores **copies** of the main memory locations that are **expectedly frequently used**.

Most modern desktop and server CPUs have at least three independent caches: the **data cache**, the **instruction cache** and the **translation look-aside buffer**.
Each location in each memory (main or cache) has
- a datum (cache line) which ranges between 8 and 512 bytes in size, while a datum requested by a CPU instruction ranges between 1 and 16.
- a unique index (called address in the case of the main memory)

In the cache, each location has also a tag (storing the address of the corresponding cached datum).
When the CPU needs to read or write a location, it checks the cache:

- if it finds it there, we have a cache hit
- if not, we have a cache miss and (in most cases) the processor needs to create a new entry in the cache.

Making room for a new entry requires a replacement policy: the Least Recently Used (LRU) discards the least recently used items first; this requires to use age bits.
CPU Cache (4/7)

Read latency (time to read a datum from the main memory) requires to keep the CPU busy with something else:

- out-of-order execution: attempt to execute independent instructions arising after the instruction that is waiting due to the cache miss

- hyper-threading (HT): allows an alternate thread to use the CPU
Modifying data in the cache requires a write policy for updating the main memory
- write-through cache: writes are immediately mirrored to main memory
- write-back cache: the main memory is mirrored when that data is evicted from the cache

The cache copy may become out-of-date or stale, if other processors modify the original entry in the main memory.
CPU Cache (6/6)

- **The replacement policy decides where in the cache a copy of a particular entry of main memory will go:**
  - **fully associative:** any entry in the cache can hold it
  - **direct mapped:** only one possible entry in the cache can hold it
  - **$N$-way set associative:** $N$ possible entries can hold it
Data locality and cache misses

Hierarchical memories and their impact on our programs

Cache Performance for SPEC CPU2000 by J.F. Cantin and M.D. Hill.

The SPEC CPU2000 suite is a collection of 26 compute-intensive, non-trivial programs used to evaluate the performance of a computer’s CPU, memory system, and compilers (http://www.spec.org/osg/cpu2000 ).
Cache issues

- **Cold miss:** The first time the data is available. Cure: Prefetching may be able to reduce this type of cost.

- **Capacity miss:** The previous access has been evicted because too much data touched in between, since the *working data set* is too large. Cure: Reorganize the data access such that *reuse* occurs before eviction.

- **Conflict miss:** Multiple data items mapped to the same location with eviction before cache is full. Cure: Rearrange data and/or pad arrays.

- **True sharing miss:** Occurs when a thread in another processor wants the same data. Cure: Minimize sharing.

- **False sharing miss:** Occurs when another processor uses different data in the same cache line. Cure: Pad data.
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;
}
```
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;
}
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(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;
}
```
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 Instructions</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>0.02</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 $L$. 

\[
A = B \times C
\]
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 $O(n^3/\sqrt{ZL})$ cache misses, if $b$ is well chosen, which is optimal.
Plan

1. Data locality and cache misses
   - Hierarchical memories and their impact on our programs
   - Cache complexity and cache-oblivious algorithms put into practice
   - A detailed case study: counting sort

2. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join multithreaded programming model
   - Anticipating parallelization overheads
   - Practical issues and optimization tricks

3. GPU programming
   - The CUDA programming and memory models

4. CUDA Programming: more details and examples

5. CUDA Programming: more details and examples
   - Tiled matrix multiplication in CUDA
   - Optimizing Matrix Transpose with CUDA
   - CUDA programming practices
The \((Z, L)\) ideal cache model (1/4)
The \((Z, L)\) ideal cache model (2/4)

- Computer with a **two-level memory hierarchy**:
  - an ideal (data) cache of \(Z\) words partitioned into \(Z/L\) cache lines, where \(L\) is the number of words per cache line.
  - an arbitrarily large main memory.
- Data moved between cache and main memory are always cache lines.
- The cache is **tall**, that is, \(Z\) is much larger than \(L\), say \(Z \in \Omega(L^2)\).
The $(Z, L)$ ideal cache model (3/4)

The processor can only reference words that reside in the cache.

If the referenced word belongs to a line already in cache, a **cache hit** occurs, and the word is delivered to the processor.

Otherwise, a **cache miss** occurs, and the line is fetched into the cache.

**Figure 1**: The ideal-cache model
The $(Z, L)$ ideal cache model (4/4)

- The ideal cache is **fully associative**: cache lines can be stored anywhere in the cache.
- The ideal cache uses the **optimal off-line strategy of replacing** the cache line whose next access is furthest in the future, and thus it exploits temporal locality perfectly.

![Diagram of the ideal-cache model](image-url)

**Figure 1**: The ideal-cache model
Cache complexity

- For an algorithm with an input of size $n$, the ideal-cache model uses two complexity measures:
  - the **work complexity** $W(n)$, which is its conventional running time in a RAM model.
  - the **cache complexity** $Q(n; Z, L)$, the number of cache misses it incurs (as a function of the size $Z$ and line length $L$ of the ideal cache).
  - When $Z$ and $L$ are clear from context, we simply write $Q(n)$ instead of $Q(n; Z, L)$.

- An algorithm is said to be **cache aware** if its behavior (and thus performances) can be tuned (and thus depend on) on the particular cache size and line length of the targeted machine.

- Otherwise the algorithm is **cache oblivious**.
Cache complexity of an array scanning

- $B$ and $N$ on the picture are our $L$ and $n$.
- Consider an array of $n$ words in main memory.
- Loading its elements by **scanning** incurs $\lceil n/L \rceil + 1$ cache misses.
- That becomes $n/L$ if $n$ divides $L$ and the array is aligned, that is, starts and ends with a cache line.
- We will often use this remark and, for simplicity, we will often replace $\lceil n/L \rceil + 1$ by $n/L$, but not always.

Figure 2. Scanning an array of $N$ elements arbitrarily aligned with blocks may cost one more memory transfer than $\lceil N/B \rceil$. 
Cache complexity of the naive and tiled matrix multiplications

- Consider square matrices of order $n$ and an $(Z,L)$-ideal cache.
- The naive multiplication (as specified before)
  
  ```java
  for(i =0; i < n; i++)
    for(j =0; j < n; j++)
      for(k=0; k < n; k++)
        C[i][j] += A[i][k] * B[k][j];
  ```

  incurs $O(n^3)$ cache misses, for $n$ large enough ($n^2 > Z$).
- The tiled multiplication (as specified before)
  
  ```java
  for(i =0; i < n/s; i++)
    for(j =0; j < n/s; j++)
      for(k=0; k < n/s; k++)
        blockMult(A,B,C,i,j,k,s);
  ```

  incurs $\Theta(n^3/(L\sqrt{Z}))$ cache misses, for $n$ large enough ($n > \sqrt{Z}$) which can be proved to be optimal, though cache-aware.
A matrix transposition cache-oblivious and cache-optimal algorithm

- Given an \( m \times n \) matrix \( A \) stored in a row-major layout, compute and store \( A^T \) into an \( n \times m \) matrix \( B \) also stored in a row-major layout.
- A naive approach would incur \( O(mn) \) cache misses, for \( n, m \) large enough.
- The algorithm \texttt{Rec-Transpose} below incurs \( \Theta(1 + mn/L) \) cache misses, which is optimal.
  1. If \( n \geq m \), the \texttt{Rec-Transpose} algorithm partitions
     \[
     A = (A_1 \ A_2), \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}
     \]
     and recursively executes \texttt{Rec-Transpose}(\( A_1, B_1 \)) and \texttt{Rec-Transpose}(\( A_2, B_2 \)).
  2. If \( m > n \), the \texttt{Rec-Transpose} algorithm partitions
     \[
     A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix}, \quad B = (B_1 \ B_2)
     \]
     and recursively executes \texttt{Rec-Transpose}(\( A_1, B_1 \)) and \texttt{Rec-Transpose}(\( A_2, B_2 \)).
void DC_matrix_transpose(int *A, int lda, int i0, int i1, int j0, int dj0, int j1 /*, int dj1 = 0 */) {
    const int THRESHOLD = 16; // tuned for the target machine
    tail:
    int di = i1 - i0, dj = j1 - j0;
    if (dj >= 2 * di && dj > THRESHOLD) {
        int dj2 = dj / 2;
        cilk_spawn DC_matrix_transpose(A, lda, i0, i1, j0, dj0, j0 + dj2);
        j0 += dj2; dj0 = 0; goto tail;
    } else if (di > THRESHOLD) {
        int di2 = di / 2;
        cilk_spawn DC_matrix_transpose(A, lda, i0, i0 + di2, j0, dj0, j1);
        i0 += di2; j0 += dj0 * di2; goto tail;
    } else {
        for (int i = i0; i < i1; ++i) {
            for (int j = j0; j < j1; ++j) {
                int x = A[j * lda + i];
                A[j * lda + i] = A[i * lda + j];
                A[i * lda + j] = x;
            }
            j0 += dj0;
        }
    }
}
Cache-oblivious matrix transposition works in practice!

<table>
<thead>
<tr>
<th>size</th>
<th>Naive</th>
<th>Cache-oblivious</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000×5000</td>
<td>126</td>
<td>79</td>
<td>1.59</td>
</tr>
<tr>
<td>10000×10000</td>
<td>627</td>
<td>311</td>
<td>2.02</td>
</tr>
<tr>
<td>20000×20000</td>
<td>4373</td>
<td>1244</td>
<td>3.52</td>
</tr>
<tr>
<td>30000×30000</td>
<td>23603</td>
<td>2734</td>
<td>8.63</td>
</tr>
<tr>
<td>40000×40000</td>
<td>62432</td>
<td>4963</td>
<td>12.58</td>
</tr>
</tbody>
</table>

- Intel(R) Xeon(R) CPU E7340 @ 2.40GHz
- L1 data 32 KB, L2 4096 KB, cache line size 64bytes
- **Both codes run on 1 core** on a node with 128GB.
- The ration comes simply from an **optimal memory access pattern**.
A cache-oblivious matrix multiplication algorithm

- To multiply an $m \times n$ matrix $A$ and an $n \times p$ matrix $B$, the Rec-Mult algorithm halves the largest of the three dimensions and recurs according to one of the following three cases:

$$
\begin{align*}
\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B &= \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}, \\
( A_1 \quad A_2 ) \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} &= A_1 B_1 + A_2 B_2, \\
A \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} &= (AB_1 \quad AB_2).
\end{align*}
\tag{1}
$$

- In case (1), we have $m \geq \max\{n, p\}$. Matrix $A$ is split horizontally, and both halves are multiplied by matrix $B$.
- In case (2), we have $n \geq \max\{m, p\}$. Both matrices are split, and the two halves are multiplied.
- In case (3), we have $p \geq \max\{m, n\}$. Matrix $B$ is split vertically, and each half is multiplied by $A$.

- The base case occurs when $m = n = p = 1$.
- The algorithm Rec-Mult above incurs

$$
\Theta(m + n + p + (mn + np + mp)/L + mnp/(L\sqrt{Z}))
$$

cache misses, which is optimal.
Plan

1. Data locality and cache misses
   - Hierarchical memories and their impact on our programs
   - Cache complexity and cache-oblivious algorithms put into practice
   - A detailed case study: counting sort

2. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join multithreaded programming model
   - Anticipating parallelization overheads
   - Practical issues and optimization tricks

3. GPU programming
   - The CUDA programming and memory models

4. CUDA Programming: more details and examples
5. CUDA Programming: more details and examples
   - Tiled matrix multiplication in CUDA
   - Optimizing Matrix Transpose with CUDA
   - CUDA programming practices
Counting sort: the algorithm

- **Counting sort** takes as input a collection of n items, each of which known by a key in the range $0 \cdots k$.
- The algorithm computes a **histogram** of the number of times each key occurs.
- Then performs a **prefix sum** to compute positions in the output.

```plaintext
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output
```
Here $n = k = 64$. Suppose $Z = 24$ and $L = 2$. Then nearly each access to Count and Output is a cache miss.
Here \( n = k = 64 \). Suppose \( Z = 24 \) and \( L = 2 \). Then nearly each access to Count and Output is a cache miss.
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1

total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output

...to compute $k$.  

1
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output

$n/L$ to compute $k$. 
allocate an array \( \text{Count}[0..k] \); initialize each array cell to zero
for each input item \( x \):
   \( \text{Count}[\text{key}(x)] = \text{Count}[\text{key}(x)] + 1 \)
\( \text{total} = 0 \)
for \( i = 0, 1, \ldots, k \):
   \( c = \text{Count}[i] \)
   \( \text{Count}[i] = \text{total} \)
   \( \text{total} = \text{total} + c \)
allocate an output array \( \text{Output}[0..n-1] \)
for each input item \( x \):
   store \( x \) in \( \text{Output}[\text{Count}[\text{key}(x)]] \)
   \( \text{Count}[\text{key}(x)] = \text{Count}[\text{key}(x)] + 1 \)
return \( \text{Output} \)

1. \( n/L \) to compute \( k \).
2. ...cache misses to initialize \( \text{Count} \).
Counting sort: cache complexity analysis (4/11)

allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output

1. \( n/L \) to compute \( k \).
2. \( k/L \) cache misses to initialize Count.
Counting sort: cache complexity analysis (5/11)

allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output

1. n/L to compute k.
2. k/L cache misses to initialize Count.
3. ... cache misses for the histogram (worst case).
Counting sort: cache complexity analysis (6/11)

allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1

total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c

allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1

return Output

1. \( n/L \) to compute \( k \).
2. \( k/L \) cache misses to initialize Count.
3. \( n/L + n \) cache misses for the histogram (worst case).
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1

total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c

allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1

return Output

1. \( n/L \) to compute \( k \).
2. \( k/L \) cache misses to initialize Count.
3. \( n/L + n \) cache misses for the histogram (worst case).
4. \( \ldots \) cache misses for the prefix sum.
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output

1. $n/L$ to compute $k$.
2. $k/L$ cache misses to initialize Count.
3. $n/L + n$ cache misses for the histogram (worst case).
4. $k/L$ cache misses for the prefix sum.
allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output

1. \(n/L\) to compute \(k\).
2. \(k/L\) cache misses to initialize Count.
3. \(n/L + n\) cache misses for the histogram (worst case).
4. \(k/L\) cache misses for the prefix sum.
5. ... cache misses for building Output (worst case).
Counting sort: cache complexity analysis (10/11)

allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ... k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output

1. \( \frac{n}{L} \) to compute \( k \).
2. \( \frac{k}{L} \) cache misses to initialize Count.
3. \( \frac{n}{L} + n \) cache misses for the histogram (worst case).
4. \( \frac{k}{L} \) cache misses for the prefix sum.
5. \( \frac{n}{L} + n + n \) cache misses for building Output (worst case).
Counting sort: cache complexity analysis (11/11)

allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
  Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, ..., k:
  c = Count[i]
  Count[i] = total
  total = total + c
allocate an output array Output[0..n-1]
for each input item x:
  store x in Output[Count[key(x)]]
  Count[key(x)] = Count[key(x)] + 1
return Output

1. \( \frac{n}{L} \) to compute \( k \).
2. \( \frac{k}{L} \) cache misses to initialize Count.
3. \( \frac{n}{L} + n \) cache misses for the histogram (worst case).
4. \( \frac{k}{L} \) cache misses for the prefix sum.
5. \( \frac{n}{L} + n + n \) cache misses for building Output (worst case).

**Total:** \( 3n + \frac{3n}{L} + 2\frac{k}{L} \) cache misses (worst case).
Counting sort: cache complexity analysis: explanations

1. \(\frac{n}{L}\) to compute \(k\): this can be done by traversing the items linearly.
2. \(\frac{k}{L}\) cache misses to initialize Count: this can be done by traversing the Count linearly.
3. \(\frac{n}{L} + n\) cache misses for the histogram (worst case): items accesses are linear but Count accesses are potentially random.
4. \(\frac{k}{L}\) cache misses for the prefix sum: Count accesses are linear.
5. \(\frac{n}{L} + n + n\) cache misses for building Output (worst case): items accesses are linear but Output and Count accesses are potentially random.

Total: \(3n + \frac{3n}{L} + \frac{2k}{L}\) cache misses (worst case).
How to fix the poor data locality of counting sort?

allocate an array Count[0..k]; initialize each array cell to zero
for each input item x:
    Count[key(x)] = Count[key(x)] + 1
total = 0
for i = 0, 1, … k:
    c = Count[i]
    Count[i] = total
    total = total + c
allocate an output array Output[0..n-1]
for each input item x:
    store x in Output[Count[key(x)]]
    Count[key(x)] = Count[key(x)] + 1
return Output

- Recall that our worst case is $3n + 3n/L + 2k/L$ cache misses.
- The troubles come from the irregular memory accesses which experience capacity misses and conflict misses.
- Workaround: we preprocess the input so that counting sort is applied in succession to several smaller input sets with smaller value ranges.
- To put it simply, so that $k$ and $n$ are small enough for Output and Count to incur cold misses only.
Counting sort: bucketing the input

allocate an array bucketsize[0..m-1]; initialize each array cell to zero
for each input item x:
    bucketsize[floor(key(x) m/(k+1))] := bucketsize[floor(key(x) m/(k+1))] + 1
total = 0
for i = 0, 1, ... m-1:
    c = bucketsize[i]
    bucketsize[i] = total
    total = total + c
allocate an array bucketedinput[0..n-1];
for each input item x:
    q := floor(key(x) m/(k+1))
        bucketedinput[bucketsize[q]] := key(x)
        bucketsize[q] := bucketsize[q] + 1
return bucketedinput

- Goal: after preprocessing, Count and Output incur **cold misses only**.
- To this end we choose a parameter $m$ (more on this later) such that
  1. a key in the range $[ih, (i + 1)h - 1]$ is always before a key in the range $[(i + 1)h, (i + 2)h - 1]$, for $i = 0 \cdots m - 2$, with $h = k/m$,
  2. bucketsize and $m$ cache-lines from bucketedinput all fit in cache. That is, counting cache-lines, $m/L + m \leq Z/L$, that is, $m + mL \leq Z$. 
Here $n = k = 64$. Suppose $Z = 24$ and $L = 2$. Then nearly each access to Count and Output is a cache miss.
Data locality and cache misses

A detailed case study: counting sort

Counting sort: bucketing illustrated (2/2)

Here \( n = k = 64 \). Suppose \( Z = 24 \) and \( L = 2 \). Then nearly each access to Count and Output is a cache miss.
Counting sort: cache complexity with bucketing

allocate an array bucketsize[0..m-1]; initialize each array cell to zero
for each input item x:
    bucketsize[floor(key(x) m/(k+1))] := bucketsize[floor(key(x) m/(k+1))] + 1
total = 0
for i = 0, 1, ... m-1:
    c = bucketsize[i]
    bucketsize[i] = total
    total = total + c
allocate an array bucketedinput[0..n-1];
for each input item x:
    q := floor(key(x) m/(k+1))
    bucketedinput[bucketsize[q] ] := key(x)
    bucketsize[q] := bucketsize[q] + 1
return bucketedinput

1. $3m/L + n/L$ caches misses to compute bucketsize
2. **Key observation:** bucketedinput is traversed regularly by segment.
3. Hence, $2n/L + m + m/L$ caches misses to compute bucketedinput

**Preprocessing:** $3n/L + 4m/L + m$ cache misses.
Counting sort: cache complexity with bucketing: explanations

1. \(3m/L + n/L\) caches misses to compute bucketsize:
   - \(m/L\) to set each cell of bucketsize to zero,
   - \(m/L + n/L\) for the first for loop,
   - \(m/L\) for the second for loop.

2. **Key observation:** bucketedinput is traversed regularly by segment:
   - So writing bucketedinput means writing (in a linear traversal) \(m\) consecutive arrays, of possibly different sizes, but with total size \(n\).
   - Thus, because of possible misalignments between those arrays and their cache-lines, this writing procedure can yield \(n/L + m\) cache misses (and not just \(n/L\)).

3. Hence, \(2n/L + m + m/L\) caches misses to compute bucketedinput:
   - \(n/L\) to read the items,
   - \(n/L + m\) to write bucketedinput,
   - \(m/L\) to load bucketsize.
Cache friendly counting sort: complete cache complexity analysis

- **Assumption:** the preprocessing creates buckets of average size $n/m$.

- After preprocessing, counting sort is applied to each bucket whose values are in a range $[ih, (i + 1)h - 1]$, for $i = 0 \cdots m - 1$, with $h = k/m$.

- To be cache-friendly, this requires, for $i = 0 \cdots m - 1$, $h + |\{\text{key} \in [ih, (i + 1)h - 1]\}| < Z$ and $m < Z/(1 + L)$. These two are very realistic assumption considering today’s cache size.

- And the total complexity becomes;

\[
Q_{\text{total}} = Q_{\text{preprocessing}} + Q_{\text{sorting}} \\
= Q_{\text{preprocessing}} + m Q_{\text{sortingofonebucket}} \\
= Q_{\text{preprocessing}} + m (3 \frac{n}{m} L + 3 \frac{n}{m} + 2 \frac{k}{mL}) \\
= Q_{\text{preprocessing}} + 6n/L + 2k/L \\
= 3n/L + 4m/L + m + 6n/L + 2k/L \\
= 9n/L + 4m/L + m + 2k/L
\]

Instead of $3n+3n/L + 2k/L$ for the naive counting sort.
Cache friendly counting sort: experimental results

- Experimentation on an Intel(R) Core(TM) i7 CPU @ 2.93GHz. It has L2 cache of 8MB.
- CPU times in seconds for both classical and cache-friendly counting sort algorithm.
- The keys are random machine integers in the range $[0, n]$.

<table>
<thead>
<tr>
<th>n</th>
<th>classical counting sort</th>
<th>cache-oblivious counting sort (preprocessing + sorting)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000000</td>
<td>13.74</td>
<td>4.66 (3.04 + 1.62)</td>
</tr>
<tr>
<td>2000000000</td>
<td>30.20</td>
<td>9.93 (6.16 + 3.77)</td>
</tr>
<tr>
<td>3000000000</td>
<td>50.19</td>
<td>16.02 (9.32 + 6.70)</td>
</tr>
<tr>
<td>4000000000</td>
<td>71.55</td>
<td>22.13 (12.50 + 9.63)</td>
</tr>
<tr>
<td>5000000000</td>
<td>94.32</td>
<td>28.37 (15.71 + 12.66)</td>
</tr>
<tr>
<td>6000000000</td>
<td>116.74</td>
<td>34.61 (18.95 + 15.66)</td>
</tr>
</tbody>
</table>
Summary and notes
# Plan

## 1. Data locality and cache misses
- Hierarchical memories and their impact on our programs
- Cache complexity and cache-oblivious algorithms put into practice
- A detailed case study: counting sort

## 2. Multicore programming
- Multicore architectures
- Cilk / Cilk++ / Cilk Plus
- The fork-join multithreaded programming model
- Anticipating parallelization overheads
- Practical issues and optimization tricks

## 3. GPU programming
- The CUDA programming and memory models

## 4. CUDA Programming: more details and examples

## 5. CUDA Programming: more details and examples
- Tiled matrix multiplication in CUDA
- Optimizing Matrix Transpose with CUDA
- CUDA programming practices
A multi-core processor is an integrated circuit to which two or more individual processors (called cores in this sense) have been attached.
Cores on a multi-core device can be coupled tightly or loosely:
  • may share or may not share a cache,
  • implement inter-core communications methods or message passing.

Cores on a multi-core implement the same architecture features as single-core systems such as instruction pipeline parallelism (ILP), vector-processing, hyper-threading, etc.
Cache Coherence (1/6)

Figure: Processor $P_1$ reads $x=3$ first from the backing store (higher-level memory)
Cache Coherence (2/6)

Figure: Next, Processor $P_2$ loads $x=3$ from the same memory
Cache Coherence (3/6)

**Figure:** Processor $P_4$ loads $x=3$ from the same memory
Figure: Processor $P_2$ issues a write $x=5$
Cache Coherence (5/6)

Figure: Processor $P_2$ writes $x=5$ in his local cache
Cache Coherence (6/6)

Figure: Processor $P_1$ issues a read $x$, which is now invalid in its cache.
MSI Protocol

- In this cache coherence protocol each block contained inside a cache can have one of three possible states:
  - **M**: the cache line has been **modified** and the corresponding data is inconsistent with the backing store; the cache has the responsibility to write the block to the backing store when it is evicted.
  - **S**: this block is unmodified and is **shared**, that is, exists in at least one cache. The cache can evict the data without writing it to the backing store.
  - **I**: this block is **invalid**, and must be fetched from memory or another cache if the block is to be stored in this cache.

- These coherency states are maintained through communication between the caches and the backing store.

- The caches have different responsibilities when blocks are read or written, or when they learn of other caches issuing reads or writes for a block.
True Sharing and False Sharing

- **True sharing:**
  - True sharing cache misses occur whenever two processors access the same data word.
  - True sharing requires the processors involved to explicitly synchronize with each other to ensure program correctness.
  - A computation is said to have **temporal locality** if it re-uses much of the data it has been accessing.
  - Programs with high temporal locality tend to have less true sharing.

- **False sharing:**
  - False sharing results when different processors use different data that happen to be co-located on the same cache line.
  - A computation is said to have **spatial locality** if it uses multiple words in a cache line before the line is displaced from the cache.
  - Enhancing spatial locality often minimizes false sharing.

See *Data and Computation Transformations for Multiprocessors* by J.M. Anderson, S.P. Amarasinghe and M.S. Lam

http://suif.stanford.edu/papers/anderson95/paper.html
Multi-core processor (cntd)

- **Advantages:**
  - Cache coherency circuitry operate at higher rate than off-chip.
  - Reduced power consumption for a dual core vs two coupled single-core processors (better quality communication signals, cache can be shared)

- **Challenges:**
  - Adjustments to existing software (including OS) are required to maximize performance
  - Production yields down (an Intel quad-core is in fact a double dual-core)
  - Two processing cores sharing the same bus and memory bandwidth may limit performances
  - High levels of false or true sharing and synchronization can easily overwhelm the advantage of parallelism
Plan

1. Data locality and cache misses
   - Hierarchical memories and their impact on our programs
   - Cache complexity and cache-oblivious algorithms put into practice
   - A detailed case study: counting sort

2. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join multithreaded programming model
   - Anticipating parallelization overheads
   - Practical issues and optimization tricks

3. GPU programming
   - The CUDA programming and memory models

4. CUDA Programming: more details and examples

5. CUDA Programming: more details and examples
   - Tiled matrix multiplication in CUDA
   - Optimizing Matrix Transpose with CUDA
   - CUDA programming practices
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 CilkPlus, see http://www.cilk.com/
- CilkPlus can be freely downloaded for Linux as a branch of the gcc compiler collection.
- Cilk is still developed at MIT http://supertech.csail.mit.edu/cilk/
Cilk++ (and Cilk Plus)

- 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

```c
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.
Loop Parallelism in CilkPlus

The iterations of a `cilk_for` loop may execute in parallel.
Cilk (resp. CilkPlus) is a multithreaded language for parallel programming that generalizes the semantics of C (resp. C++) by introducing linguistic constructs for parallel control.

Cilk (resp. CilkPlus) is a **faithful extension** of C (resp. C++):

- The C (resp. C++) elision of a Cilk (resp. CilkPlus) is a correct implementation of the semantics of the program.
- Moreover, on one processor, a parallel Cilk (resp. CilkPlus) program scales down to run nearly as fast as its C (resp. C++) elision.

To obtain the serialization of a CilkPlus program

```c
#define cilk_for for
#define cilk_spawn
#define cilk_sync
```
Multicore programming

Serial Semantics (2/2)

```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);
    }
}
```

Cilk++ source

Serialization
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*.
- CilkPlus’s scheduler maps strands onto processors dynamically at runtime.
The CilkPlus Platform

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);
    }
}

Cilk Plus
Compiler
Conventional
Compiler
Hyperobject
Library
Cilkview
Scalability Analyzer
Cilkscreen
Race Detector
Binary
Linker
Runtime System
Reliable Single-
Threaded Code
Exceptional
Performance
Reliable Multi-
Threaded Code
Parallel Regression Tests
Conventional Regression Tests
Serialization

Cilk++ source
Benchmarks for the parallel version of the cache-oblivious mm

Multiplying a 4000x8000 matrix by a 8000x4000 matrix

- on 32 cores = 8 sockets × 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>
So does the (tuned) cache-oblivious matrix multiplication
Plan

1. Data locality and cache misses
   - Hierarchical memories and their impact on our programs
   - Cache complexity and cache-oblivious algorithms put into practice
   - A detailed case study: counting sort

2. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join multithreaded programming model
   - Anticipating parallelization overheads
   - Practical issues and optimization tricks

3. GPU programming
   - The CUDA programming and memory models

4. CUDA Programming: more details and examples

5. CUDA Programming: more details and examples
   - Tiled matrix multiplication in CUDA
   - Optimizing Matrix Transpose with CUDA
   - CUDA programming practices
The fork-join parallelism model

```
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);
  }
}
```

Example:
```
fib(4)
```

"Processor oblivious"

The computation dag unfolds dynamically.
The fork-join parallelism model

Figure: Instruction stream DAG.

\( T_p \) is the minimum running time on \( p \) processors.

\( T_1 \) is the sum of the number of instructions at each vertex in the DAG, called the **work**.

\( T_\infty \) is the minimum running time with infinitely many processors, called the **span**. This is the length of a path of maximum length from the root to a leaf.

\( T_1/T_\infty \): **Parallelism**.

- **Work law**: \( T_p \geq T_1/p \).
- **Span law**: \( T_p \geq T_\infty \).
For **loop parallelism in Cilk++**

\[
\begin{pmatrix}
a_{11} & a_{12} & \cdots & a_{1n} \\
a_{21} & a_{22} & \cdots & a_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nn}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
a_{11} & a_{21} & \cdots & a_{n1} \\
a_{12} & a_{22} & \cdots & a_{n2} \\
\vdots & \vdots & \ddots & \vdots \\
a_{1n} & a_{2n} & \cdots & a_{nn}
\end{pmatrix}
\]

\[
A \rightarrow A^T
\]

cilk_for (int i=1; i<n; ++i) {
    for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}

The iterations of a cilk_for loop execute in parallel.
Implementation of for loops in Cilk++

Up to details (next week!) the previous loop is compiled as follows, using a divide-and-conquer implementation:

```c
void recur(int lo, int hi) {
    if (hi > lo) { // coarsen
        int mid = lo + (hi - lo)/2;
        cilk_spawn recur(lo, mid);
        recur(mid+1, hi);
        cilk_sync;
    } else
        for (int j=0; j<hi; ++j) {
        double temp = A[hi][j];
        A[hi][j] = A[j][hi];
        A[j][hi] = temp;
        }
}
```
For loops in the fork-join parallelism model: another example

cilk_for (int i = 1; i <= 8; i ++){
    f(i);
}

A \texttt{cilk\_for} loop executes recursively as 2 for loops of $n/2$ iterations, adding a span of $\Theta(\log(n))$.

\textbf{Figure}: DAG for a \texttt{cilk\_for} with 8 iterations.
The work-stealing scheduler (1/11)
The work-stealing scheduler (2/11)
The work-stealing scheduler (3/11)
The work-stealing scheduler (4/11)
The work-stealing scheduler (5/11)
The work-stealing scheduler (6/11)
The work-stealing scheduler (7/11)
The work-stealing scheduler (8/11)
The work-stealing scheduler (9/11)
The work-stealing scheduler (10/11)
The work-stealing scheduler (11/11)
Performances of the work-stealing scheduler

Assume that

- each strand executes in unit time,
- for almost all “parallel steps” there are at least $p$ strands to run,
- each processor is either working or stealing.

Then, the randomized work-stealing scheduler is expected to run in

$$T_P = T_1/p + O(T_\infty)$$
Overheads and burden

- Many factors (simplification assumptions of the fork-join parallelism model, architecture limitation, costs of executing the parallel constructs, overheads of scheduling) will make $T_p$ larger in practice than $T_1/p + T_\infty$.

- Cilk++ estimates $T_p$ as $T_p = T_1/p + 1.7 \text{ burden\_span}$, where burden\_span is 15000 instructions times the number of continuation edges along the critical path.
Cilkview computes work and span to derive upper bounds on parallel performance.

Cilkview also estimates scheduling overhead to compute a burdened span for lower bounds.
Tuned cache-oblivious parallel matrix multiplication
Plan

1. Data locality and cache misses
   - Hierarchical memories and their impact on our programs
   - Cache complexity and cache-oblivious algorithms put into practice
   - A detailed case study: counting sort

2. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join multithreaded programming model
   - Anticipating parallelization overheads
   - Practical issues and optimization tricks

3. GPU programming
   - The CUDA programming and memory models

4. CUDA Programming: more details and examples
5. CUDA Programming: more details and examples
   - Tiled matrix multiplication in CUDA
   - Optimizing Matrix Transpose with CUDA
   - CUDA programming practices
Construction of the Pascal Triangle: nearly the simplest stencil computation!
Divide and conquer: principle

The parallelism is $\Theta(n^{2-\log_2 3})$, so roughly $\Theta(n^{0.45})$ which can be regarded as low parallelism.
Let $B$ be the order of a block and $n$ be the number of elements.

The parallelism of $\Theta(n/B)$ can still be regarded as low parallelism, but better than with the divide and conquer scheme.
Estimating parallelization overheads

The instruction stream DAG of the blocking strategy consists of \( n/B \) binary trees \( T_0, T_1, \ldots, T_{n/B-1} \) such that

- \( T_i \) is the instruction stream DAG of the \texttt{cilk for} loop executing the \( i \)-th band
- each leaf of \( T_i \) is connected by an edge to the root of \( T_{i+1} \).

Consequently, the burdened span is

\[
S_b(n) = \sum_{i=1}^{n/B} \log(i) = \log\left( \prod_{i=1}^{n/B} i \right) = \log(\Gamma\left( \frac{n}{B} + 1 \right)).
\]

Using Stirling’s Formula, we deduce

\[
S_b(n) \in \Theta\left( \frac{n}{B} \log\left( \frac{n}{B} \right) \right). \tag{4}
\]

Thus the burdened parallelism (that is, the ratio work to burdened span) is \( \Theta(Bn/log(\frac{n}{B})) \), that is sub-linear in \( n \), while the non-burdened parallelism is \( \Theta(n/B) \).
Construction of the Pascal Triangle: experimental results

Worker vs Speedup and Parallelism

- speedup dynamic block
- speedup static block
- parallelism dynamic block
- parallelism static block

Core/Workers vs Speedup and Parallelism graph
Summary and notes

Burdened parallelism

- Parallelism after accounting for parallelization overheads (thread management, costs of scheduling, etc.) The **burdened parallelism** is estimated as the ratio work to burdened span.
- The **burdened span** is defined as the maximum number of spawns/syncs on a critical path times the cost for a `cilk_spawn` (`cilk_sync`) taken as 15,000 cycles.

Impact in practice: example for the Pascal Triangle

- Consider executing one band after another, where for each band all $B \times B$ blocks are executed concurrently.
- The **non-burdened span** is in $\Theta(B^2n/B) = \Theta(n/B)$.
- While the **burdened span** is

$$S_b(n) = \sum_{i=1}^{n/B} \log(i) = \log(\prod_{i=1}^{n/B} i) = \log(\Gamma\left(\frac{n}{B} + 1\right)) \in \Theta\left(\frac{n}{B} \log\left(\frac{n}{B}\right)\right).$$
Plan

1. Data locality and cache misses
   - Hierarchical memories and their impact on our programs
   - Cache complexity and cache-oblivious algorithms put into practice
   - A detailed case study: counting sort

2. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join multithreaded programming model
   - Anticipating parallelization overheads
   - Practical issues and optimization tricks

3. GPU programming
   - The CUDA programming and memory models

4. CUDA Programming: more details and examples

5. CUDA Programming: more details and examples
   - Tiled matrix multiplication in CUDA
   - Optimizing Matrix Transpose with CUDA
   - CUDA programming practices
Example 1: a small loop with grain size = 1

Code:

```c
const int N = 100 * 1000 * 1000;

void cilk_for_grainsize_1()
{
    #pragma cilk_grainsize = 1
    cilk_for (int i = 0; i < N; ++i)
        fib(2);
}
```

Expectations:
- Parallelism should be large, perhaps $\Theta(N)$ or $\Theta(N/\log N)$.
- We should see great speedup.
Speedup is indeed great...
...but performance is lousy
Recall how \texttt{cilk\_for} is implemented

Source:

\begin{verbatim}
cilk_for (int i = A; i < B; ++i)
\hspace{1em}BODY(i)
\end{verbatim}

Implementation:

\begin{verbatim}
void recur(int lo, int hi) {
    if ((hi - lo) > GRAINSIZE) {
        int mid = lo + (hi - lo) / 2;
        cilk_spawn recur(lo, mid);
        cilk_spawn recur(mid, hi);
    } else
        for (int i = lo; i < hi; ++i)
            BODY(i);
}

recur(A, B);
\end{verbatim}
Default grain size

Cilk++ chooses a grain size if you don’t specify one.

```c
void cilk_for_default_grainsize()
{
    cilk_for (int i = 0; i < N; ++i)
        fib(2);
}
```

Cilk++’s heuristic for the grain size:

\[
\text{grain size} = \min \left\{ \frac{N}{8P}, 512 \right\}.
\]

- Generates about $8P$ parallel leaves.
- Works well if the loop iterations are not too unbalanced.
Speedup with default grain size
Large grain size

A large grain size should be even faster, right?

```c
void cilk_for_large_grainsize()
{
    #pragma cilk_grainsize = N
    cilk_for (int i = 0; i < N; ++i)
        fib(2);
}
```

Actually, no (except for noise):

<table>
<thead>
<tr>
<th>Grain size</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.55 s</td>
</tr>
<tr>
<td>default (= 512)</td>
<td>2.44 s</td>
</tr>
<tr>
<td>$N$ (= $10^8$)</td>
<td>2.42 s</td>
</tr>
</tbody>
</table>
Speedup with grain size $= N$
Trade-off between grain size and parallelism

Use Cilkview to understand the trade-off:

<table>
<thead>
<tr>
<th>Grain size</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 default (= 512)</td>
<td>6,951,154</td>
</tr>
<tr>
<td>(N (= 10^8))</td>
<td>248,784</td>
</tr>
</tbody>
</table>

In Cilkview, \(P = 1\):

\[
\text{default grain size} = \min \left\{ \frac{N}{8P}, 512 \right\} = \min \left\{ \frac{N}{8}, 512 \right\}.
\]
Lessons learned

- Measure overhead before measuring speedup.
  - Compare 1-processor Cilk++ versus serial code.
- Small grain size $\Rightarrow$ higher work overhead.
- Large grain size $\Rightarrow$ less parallelism.
- The default grain size is designed for small loops that are reasonably balanced.
  - You may want to use a smaller grain size for unbalanced loops or loops with large bodies.
- Use Cilkview to measure the parallelism of your program.
Example 2: A `for` loop that spawns

Code:

```c
const int N = 10 * 1000 * 1000;

/* empty test function */
void f() {
}

void for_spawn()
{
    for (int i = 0; i < N; ++i)
        cilk_spawn f();
}
```

Expectations:
- I am spawning $N$ parallel things.
- Parallelism should be $\Theta(N)$, right?
“Speedup” of for_spawn()
Insufficient parallelism

PPA analysis:

- PPA says that both work and span are $\Theta(N)$.
- Parallelism is $\approx 1.62$, independent of $N$.
- Too little parallelism: no speedup.

Why is the span $\Theta(N)$?

```c
for (int i = 0; i < N; ++i)
    cilk_spawn f();
```
Alternative: a cilk_for loop.

Code:

```c
/* empty test function */
void f() { }

void test_cilk_for()
{
    cilk_for (int i = 0; i < N; ++i)
        f();
}
```

PPA analysis:

The parallelism is about 2000 (with default grain size).

- The parallelism is high.
- As we saw earlier, this kind of loop yields good performance and speedup.
Lessons learned

- `cilk_for()` is different from `for(...) cilk_spawn`.
- The span of `for(...) cilk_spawn` is $\Omega(N)$.
- For simple flat loops, `cilk_for()` is generally preferable because it has higher parallelism.
- (However, `for(...) cilk_spawn` might be better for recursively nested loops.)
- Use Cilkview to measure the parallelism of your program.
Example 3: Vector addition

Code:

```c
const int N = 50 * 1000 * 1000;

double A[N], B[N], C[N];

void vector_add()
{
    cilk_for (int i = 0; i < N; ++i)
        A[i] = B[i] + C[i];
}
```

Expectations:
- Cilkview says that the parallelism is 68,377.
- This will work great!
Speedup of `vector_add()`
Bandwidth of the memory system

A typical machine: AMD Phenom 920 (Feb. 2009).

<table>
<thead>
<tr>
<th>Cache level</th>
<th>daxpy bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>19.6 GB/s per core</td>
</tr>
<tr>
<td>L2</td>
<td>18.3 GB/s per core</td>
</tr>
<tr>
<td>L3</td>
<td>13.8 GB/s shared</td>
</tr>
<tr>
<td>DRAM</td>
<td>7.1 GB/s shared</td>
</tr>
</tbody>
</table>

**daxpy**: $x[i] = a \times x[i] + y[i]$, double precision.

The memory bottleneck:

- A single core can generally saturate most of the memory hierarchy.
- Multiple cores that access memory will conflict and slow each other down.
How do you determine if memory is a bottleneck?

Hard problem:
- No general solution.
- Requires guesswork.

Two useful techniques:
- Use a profiler such as the Intel VTune.
  - Interpreting the output is nontrivial.
  - No sensitivity analysis.
- Perturb the environment to understand the effect of the CPU and memory speeds upon the program speed.
How to perturb the environment

- Overclock/underclock the processor, e.g. using the power controls.
  - If the program runs at the same speed on a slower processor, then the memory is (probably) a bottleneck.
- Overclock/underclock the DRAM from the BIOS.
  - If the program runs at the same speed on a slower DRAM, then the memory is not a bottleneck.
- Add spurious work to your program while keeping the memory accesses constant.
- Run $P$ independent copies of the serial program concurrently.
  - If they slow each other down then memory is probably a bottleneck.
Perturbing `vector_add()`

```c
const int N = 50 * 1000 * 1000;

double A[N], B[N], C[N];

void vector_add()
{
    cilk_for (int i = 0; i < N; ++i) {
        A[i] = B[i] + C[i];
        fib(5);  // waste time
    }
}
```
Speedup of perturbed \texttt{vector\_add()}

![Graph showing speedup of perturbed \texttt{vector\_add()} for different numbers of processors.](image)
Interpreting the perturbed results

The memory is a bottleneck:

- A little extra work ($\text{fib}(5)$) keeps 8 cores busy. A little more extra work ($\text{fib}(10)$) keeps 16 cores busy.
- Thus, we have enough parallelism.
- The memory is *probably* a bottleneck. (If the machine had a shared FPU, the FPU could also be a bottleneck.)

OK, but how do you fix it?

- `vector_add` cannot be fixed in isolation.
- You must generally restructure your program to increase the reuse of cached data. Compare the iterative and recursive matrix multiplication from yesterday.
- (Or you can buy a newer CPU and faster memory.)
Lessons learned

- Memory is a common bottleneck.
- One way to diagnose bottlenecks is to perturb the program or the environment.
- Fixing memory bottlenecks usually requires algorithmic changes.
Example 4: Nested loops

Code:

```c
const int N = 1000 * 1000;

void inner_parallel()
{
    for (int i = 0; i < N; ++i)
        cilk_for (int j = 0; j < 4; ++j)
            fib(10); /* do some work */
}
```

Expectations:
- The inner loop does 4 things in parallel. The parallelism should be about 4.
- Cilkview says that the parallelism is 3.6.
- We should see some speedup.
“Speedup” of `inner_parallell()`
Interchanging loops

Code:

```c
const int N = 1000 * 1000;

void outer_parallel()
{
    cilk_for (int j = 0; j < 4; ++j)
        for (int i = 0; i < N; ++i)
            fib(10); /* do some work */
}
```

Expectations:
- The outer loop does 4 things in parallel. The parallelism should be about 4.
- Cilkview says that the parallelism is 4.
- Same as the previous program, which didn’t work.
Speedup of `outer_parallel()`
**Parallelism vs. burdened parallelism**

**Parallelism:**
The best speedup you can hope for.

**Burdened parallelism:**
Parallelism after accounting for the unavoidable migration overheads.

Depends upon:
- How well we implement the Cilk++ scheduler.
- How you express the parallelism in your program.

**Cilkview prints the burdened parallelism:**
- 0.29 for `inner_parallel()`, 4.0 for `outer_parallel()`.
- In a good program, parallelism and burdened parallelism are about equal.
What is the burdened parallelism?

Code:

A();
cilk_spawn B();
C();
D();
cilk_sync;
E();

Burdened critical path:

![Control flow graph of the code](image)

The **burden** is $\Theta(10000)$ cycles (locks, malloc, cache warmup, reducers, etc.)
The burden in our examples

$\Theta(N)$ spawns/syncs on the critical path (large burden):

```c
void inner_parallel()
{
    for (int i = 0; i < N; ++i)
        cilk_for (int j = 0; j < 4; ++j)
            fib(10); /* do some work */
}
```

$\Theta(1)$ spawns/syncs on the critical path (small burden):

```c
void outer_parallel()
{
    cilk_for (int j = 0; j < 4; ++j)
        for (int i = 0; i < N; ++i)
            fib(10); /* do some work */
}
Lessons learned

- Insufficient parallelism yields *no speedup*; high burden yields *slowdown*.
- Many spawns but small parallelism: suspect large burden.
- Cilkview helps by printing the burdened span and parallelism.
- The burden can be interpreted as the number of spawns/syncs on the critical path.
- If the burdened parallelism and the parallelism are approximately equal, your program is ok.
Summary and notes

We have learned to identify and (when possible) address these problems:

- High overhead due to small grain size in cilk_for loops.
- Insufficient parallelism.
- Insufficient memory bandwidth.
- Insufficient burdened parallelism.
Plan

1. Data locality and cache misses
   - Hierarchical memories and their impact on our programs
   - Cache complexity and cache-oblivious algorithms put into practice
   - A detailed case study: counting sort

2. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join multithreaded programming model
   - Anticipating parallelization overheads
   - Practical issues and optimization tricks

3. GPU programming
   - The CUDA programming and memory models

4. CUDA Programming: more details and examples

5. CUDA Programming: more details and examples
   - Tiled matrix multiplication in CUDA
   - Optimizing Matrix Transpose with CUDA
   - CUDA programming practices
CUDA design goals

- Enable heterogeneous systems (i.e., CPU+GPU)
- Scale to 100’s of cores, 1000’s of parallel threads
- Use C/C++ with minimal extensions
- Let programmers focus on parallel algorithms
Heterogeneous programming (1/3)

- A CUDA program is a serial program with parallel kernels, all in C.
- The serial C code executes in a host (= CPU) thread
- The parallel kernel C code executes in many device threads across multiple GPU processing elements, called streaming processors (SP).
Heterogeneous programming (2/3)

- Thus, the parallel code (kernel) is launched and executed on a device by many threads.
- Threads are grouped into thread blocks.
- One kernel is executed at a time on the device.
- Many threads execute each kernel.
Heterogeneous programming (3/3)

- The parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in **thread and block ID variables** are used to map each thread to a specific data tile (see next slide).

- Thus, each thread executes the same code on different data based on its thread and block ID.
Example: increment array elements (1/2)

Increment N-element vector \( a \) by scalar \( b \)

Let’s assume \( N=16 \), \( \text{blockDim}=4 \) \( \rightarrow \) 4 blocks

\[
\text{int idx} = \text{blockDim.x} \times \text{blockIdx.x} + \text{threadIdx.x};
\]

- \( \text{blockIdx.x}=0 \), \( \text{blockDim.x}=4 \)
  - \( \text{threadIdx.x}=0,1,2,3 \)
  - \( \text{idx}=0,1,2,3 \)
- \( \text{blockIdx.x}=1 \), \( \text{blockDim.x}=4 \)
  - \( \text{threadIdx.x}=0,1,2,3 \)
  - \( \text{idx}=4,5,6,7 \)
- \( \text{blockIdx.x}=2 \), \( \text{blockDim.x}=4 \)
  - \( \text{threadIdx.x}=0,1,2,3 \)
  - \( \text{idx}=8,9,10,11 \)
- \( \text{blockIdx.x}=3 \), \( \text{blockDim.x}=4 \)
  - \( \text{threadIdx.x}=0,1,2,3 \)
  - \( \text{idx}=12,13,14,15 \)

See our example number 4 in /usr/local/cs4402/examples/4
Example: increment array elements (2/2)

**CPU program**

```c
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx < N; idx++)
        a[idx] = a[idx] + b;
}

void main()
{
    ......  
    increment_cpu(a, b, N);
}
```

**CUDA program**

```c
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}

void main()
{
    ......  
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize) );
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```
Example host code for increment array elements

```c
// allocate host memory
unsigned int numBytes = N * sizeof(float);
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**)&d_A, numBytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);

// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
```
A **Thread block** is a group of threads that can:
- Synchronize their execution
- Communicate via shared memory

Within a grid, **thread blocks can run in any order**:
- Concurrently or sequentially
- Facilitates scaling of the same code across many devices
Thus, within a grid, any possible interleaving of blocks must be valid.

Thread blocks **may coordinate but not synchronize**
- they may share pointers
- they should not share locks (this can easily deadlock).

The fact that thread blocks cannot synchronize gives **scalability**:
- A kernel scales across any number of parallel cores

However, within a thread block, threads may synchronize with barriers.

That is, threads wait at the barrier until **all** threads in the **same block** reach the barrier.
Vector addition on GPU (1/4)

```c
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```
Vector addition on GPU (2/4)

```c
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
}
```
Vector addition on GPU (3/4)

```cpp
// allocate and initialize host (CPU) memory
float *h_A = ...;  *h_B = ...;  *h_C = ... (empty)

// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
cudaMalloc((void**) &d_A, N * sizeof(float));
cudaMalloc((void**) &d_B, N * sizeof(float));
cudaMalloc((void**) &d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy(d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice);

// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```
// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);

// copy result back to host memory

cudaMemcpy(h_C, d_C, N * sizeof(float), cudaMemcpyDeviceToHost);

// do something with the result...

// free device (GPU) memory

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
Memory hierarchy (1/3)

Host (CPU) memory:
- Not directly accessible by CUDA threads

```
cudaMemcpy()
```
Memory hierarchy (2/3)

**Global (on the device) memory:**
- Also called **device memory**
- Accessible by all threads as well as host (CPU)
- Data lifetime = from allocation to deallocation
Memory hierarchy (3/3)

**Shared memory:**
- Each thread block has its own shared memory, which is accessible only by the threads within that block
- Data lifetime = block lifetime

**Local storage:**
- Each thread has its own local storage
- Data lifetime = thread lifetime
Blocks run on multiprocessors

Kernel launched by host

Device processor array

Device Memory
Streaming processors and multiprocessors
Hardware multithreading

- **Hardware allocates resources to blocks:**
  - blocks need: thread slots, registers, shared memory
  - blocks don’t run until resources are available

- **Hardware schedules threads:**
  - threads have their own registers
  - any thread not waiting for something can run
  - context switching is free every cycle

- **Hardware relies on threads to hide latency:**
  - thus high parallelism is necessary for performance.
**SIMT thread execution**

- At each clock cycle, a multiprocessor executes the same instruction on a group of threads called a **warp**
  - The number of threads in a warp is the **warp size** (32 on G80)
  - A half-warp is the first or second half of a warp.
- Within a warp, threads
  - share instruction fetch/dispatch
  - some become inactive when code path diverges
  - hardware automatically handles divergence
- **Warps are the primitive unit of scheduling**:
  - each active block is split into warps in a well-defined way
  - threads within a warp are executed physically in parallel while warps and blocks are executed logically in parallel.
Code executed on the GPU

- The GPU code defines and calls C function with some restrictions:
  - Can only access GPU memory
  - No variable number of arguments
  - No static variables
  - No recursion (...well this has changed recently)
  - No dynamic polymorphism

- GPU functions must be declared with a qualifier:
  - `__global__`: launched by CPU, cannot be called from GPU, must return void
  - `__device__`: called from other GPU functions, cannot be launched by the CPU
  - `__host__`: can be executed by CPU

- Qualifiers can be combined.

- Built-in variables: `gridDim`, `blockDim`, `blockIdx`, `threadIdx`
Variable Qualifiers (GPU code)

__device__:  
- stored in global memory (not cached, high latency)
- accessible by all threads
- lifetime: application

__constant__:  
- stored in global memory (cached)
- read-only for threads, written by host
- Lifetime: application

__shared__:  
- stored in shared memory (latency comparable to registers)
- accessible by all threads in the same threadblock
- lifetime: block lifetime

Unqualified variables:  
- scalars and built-in vector types are stored in registers
- arrays are stored in device (≡ global) memory
Launching kernels on GPU

Launch parameters:

- grid dimensions (up to 2D)
- thread-block dimensions (up to 3D)
- shared memory: number of bytes per block
  - for extern smem variables declared without size
  - Optional, 0 by default
- stream ID:
  - Optional, 0 by default

```c
dim3 grid(16, 16);
dim3 block(16,16);
kernell<<<grid, block, 0, 0>>>(...);
kernell<<<32, 512>>>(...);
```
Host (CPU) manages GPU memory:

- `cudaMalloc (void ** pointer, size_t nbytes)`
- `cudaMemset (void * pointer, int value, size_t count)`
- `cudaFree (void* pointer)`

```c
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**) &d_a, nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```
CUDA Programming: more details and examples

Data Copies

- cudaMemcpy( void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  - returns after the copy is complete,
  - blocks the CPU thread,
  - doesn’t start copying until previous CUDA calls complete.

- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

- Non-blocking memcopies are provided (more on this later)
Thread Synchronization Function

- `void __syncthreads();`

- Synchronizes all threads in a block:
  - once all threads have reached this point, execution resumes normally.
  - this is used to avoid hazards when accessing shared memory.

- Should be used in conditional code only if the condition is uniform across the entire thread block.
Example kernel Source Code

```c
__global__ void sum_kernel(int *g_input, int *g_output)
{
    extern __shared__ int s_data[ ]; // allocated during kernel launch

    // read input into shared memory
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    s_data[ threadIdx.x ] = g_input[ idx ];
    __syncthreads( );

    // compute sum for the threadblock
    for ( int dist = blockDim.x/2; dist > 0; dist /= 2 )
    {
        if ( threadIdx.x < dist )
            s_data[ threadIdx.x ] += s_data[ threadIdx.x + dist ];
        __syncthreads( );
    }

    // write the block's sum to global memory
    if ( threadIdx.x == 0 )
        g_output[ blockIdx.x ] = s_data[0];
}
```

Kernel variations and output: what is in \( a \)?

```c
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}

__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}

__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```
**Kernel variations and output: answers**

```c
__global__ void kernel(int *a) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
Output: 7 7 7 7 7 7 7 7 7 7 7 7

__global__ void kernel(int *a) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
Output: 0 0 0 0 1 1 1 2 2 2 2 3 3 3

__global__ void kernel(int *a) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
```
Code Walkthrough (1/4)

// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a = 0, *h_a = 0; // device and host pointers
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers
    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemcpy( d_a, 0, num_bytes );
    cudaMemcpyDeviceToHost();
```c
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers
    h_a = (int*)malloc(num_bytes);
    cudaMemcpy( (void**)d_a, num_bytes);
    cudaMemcpy( (void**)d_a, num_bytes);

    if( 0==h_a || 0==d_a )
    {
        printf("can't allocate memory\n");
        return 1;
    }

    cudaMemcpy( d_a, 0, num_bytes);
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost);

    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i]);
    printf("\n");

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```
Example: Shuffling Data

```c
// Reorder values based on keys
// Each thread moves one element
__global__ void shuffle(int* prev_array, int* new_array, int* indices)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    new_array[i] = prev_array[indices[i]];
}
```

```
int main()
{
    // Run grid of N/256 blocks of 256 threads each
    shuffle<<<N/256, 256>>>(d_old, d_new, d_ind);
}
```
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix = blockIdx.x*blockDim.x + threadIdx.x;
    int iy = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx] = a[idx]+1;
}
```c
int main()
{
    int dimx = 16;
    int dimy = 16;
    int num_bytes = dimx*dimy*sizeof(int);

    int *d_a = 0, *h_a = 0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMemcpy( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemcpy( d_a, 0, num_bytes );

    dim3 grid, block;
    block.x = 4;
    block.y = 4;
    grid.x = dimx / block.x;
    grid.y = dimy / block.y;

    kernel<<<grid, block>>>( d_a, dimx, dimy );

    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int row=0; row<dimy; row++)
    {
        for(int col=0; col<dimx; col++)
            printf("%d ", h_a[row*dimx+col]);
        printf("\n");
    }

    free( h_a );
    cudaFree( d_a );

    return 0;
}
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix = blockIdx.x*blockDim.x + threadIdx.x;
    int iy = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx] = a[idx]+1;
}
```
Plan

1. Data locality and cache misses
   - Hierarchical memories and their impact on our programs
   - Cache complexity and cache-oblivious algorithms put into practice
   - A detailed case study: counting sort

2. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join multithreaded programming model
   - Anticipating parallelization overheads
   - Practical issues and optimization tricks

3. GPU programming
   - The CUDA programming and memory models

4. CUDA Programming: more details and examples

5. CUDA Programming: more details and examples
   - Tiled matrix multiplication in CUDA
   - Optimizing Matrix Transpose with CUDA
   - CUDA programming practices
The goals of this example are:

- Understanding how to write a kernel for a non-toy example
- Understanding how to map work (and data) to the thread blocks
- Understanding the importance of using shared memory

We start by writing a naive kernel for matrix multiplication which does not use shared memory.

Then we analyze the performance of this kernel and realize that it is limited by the global memory latency.

Finally, we present a more efficient kernel, which takes advantage of a tile decomposition and makes use of shared memory.
Matrix multiplication (2/16)

- Consider multiplying two rectangular matrices $A$ and $B$ with respective formats $m \times n$ and $n \times p$. Define $C = A \times B$.
- Principle: each thread computes an element of $C$ through a 2D grid with 2D thread blocks.
Matrix multiplication (3/16)

```c
__global__ void mat_mul(float *a, float *b,
                        float *ab, int width)
{
    // calculate the row & col index of the element
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col = blockIdx.x*blockDim.x + threadIdx.x;
    float result = 0;
    // do dot product between row of a and col of b
    for(int k = 0; k < width; ++k)
        result += a[row*width+k] * b[k*width+col];
    ab[row*width+col] = result;
}
```
Matrix multiplication (4/16)

- Analyze the previous CUDA kernel for multiplying two rectangular matrices $A$ and $B$ with respective formats $m \times n$ and $n \times p$. Define $C = A \times B$.
- Each element of $C$ is computed by one thread:
  - then each row of $A$ is read $p$ times and
  - each column of $B$ is read $m$ times, thus
  - $2mnp$ reads in total for $2mnp$ flops.
- Let $t$ be an integer dividing $m$ and $p$. We decompose $C$ into $t \times t$ tiles. If tiles are computed one after another, then:
  - $(m/t)(tn)(p/t)$ slots are read in $A$
  - $(p/t)(tn)(m/t)$ slots are read in $B$, thus
  - $2mnp/t$ reads in total for $2mnp$ flops.
- For a CUDA implementation, $t = 16$ such that each tile is computed by one thread block.
The previous explanation can be adapted to a particular GPU architecture, so as to estimate the performance of the first (naive) kernel.

The first kernel has a global memory access to flop ratio (GMAC) of 8 Bytes / 2 ops, that is, 4 B/op.

Suppose using a GeForce GTX 260, which has 805 GFLOPS peak performance.

In order to reach peak fp performance we would need a memory bandwidth of GMAC × Peak FLOPS = 3.2 TB/s.

Unfortunately, we only have 112 GB/s of actual memory bandwidth (BW) on a GeForce GTX 260.

Therefore an upper bound on the performance of our implementation is BW / GMAC = 28 GFLOPS.
Matrix multiplication (6/16)

- The picture below illustrates our second kernel.
- Each thread block computes a tile in $C$, which is obtained as a dot product of tile-vector of $A$ by a tile-vector of $B$.
- Tile size is chosen in order to maximize data locality.
Matrix multiplication (7/16)

- So a thread block computes a $t \times t$ tile of $C$.
- Each element in that tile is a dot-product of a row from $A$ and a column from $B$.
- We view each of these dot-products as a sum of small dot products:

$$c_{i,j} = \sum_{k=o}^{t-1} a_{i,k} b_{k,j} + \sum_{k=t}^{2t-1} a_{i,k} b_{k,j} + \cdots + \sum_{k=n-1-t}^{n-1} a_{i,k} b_{k,j}$$

- Therefore we fix $\ell$ and then compute $\sum_{k=\ell t}^{(\ell+1)t-1} a_{i,k} b_{k,j}$ for all $i,j$ in the working thread block.
- We do this for $\ell = 0, 1, \ldots, (n/t - 1)$.
- This allows us to store the working tiles of $A$ and $B$ in shared memory.
Matrix multiplication (8/16)

- We assume that $A$, $B$, $C$ are stored in row-major layout.
- Observe that for computing a tile in $C$ our kernel code does need to know the number of rows in $A$.
- It just needs to know the width (number of columns) of $A$ and $B$.

```c
#define BLOCK_SIZE 16

template <typename T>
__global__ void matrix_mul_ker(T* C, const T *A, const T *B,
    size_t wa, size_t wb)

    // Block index; WARNING: should be at most 2^16 - 1
    int bx = blockIdx.x; int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x; int ty = threadIdx.y;
```
Matrix multiplication (9/16)

- We need the position in *A of the first element of the first working tile from $A$; we call it $aBegin$.
- We will need also the position in *A of the last element of the first working tile from $A$; we call it $aEnd$.
- Moreover, we will need the offset between two consecutive working tiles of $A$; we call it $aStep$.

```c
int aBegin = wa * BLOCK_SIZE * by;
int aEnd = aBegin + wa - 1;
int aStep = BLOCK_SIZE;
```
Similarly for $B$ we have $b\text{Begin}$ and $b\text{Step}$.

We will not need a $b\text{End}$ since once we are done with a row of $A$, we are also done with a column of $B$.

Finally, we initialize the accumulator of the working thread; we call it $C_{\text{Sub}}$.

```cpp
int b\text{Begin} = \text{BLOCK\_SIZE} * bx;

int b\text{Step} = \text{BLOCK\_SIZE} * wb;

int C_{\text{Sub}} = 0;
```
The main loop starts by copying the working tiles of $A$ and $B$ to shared memory.

```cpp
for(int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
{
    // shared memory for the tile of A
    __shared__ int As[BLOCK_SIZE][BLOCK_SIZE];

    // shared memory for the tile of B
    __shared__ int Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load the tiles from global memory to shared memory
    // each thread loads one element of each tile
    As[ty][tx] = A[a + wa * ty + tx];
    Bs[ty][tx] = B[b + wb * ty + tx];

    // synchronize to make sure the matrices are loaded
    __syncthreads();
```
Compute a small “dot-product” for each element in the working tile of $C$.

// Multiply the two tiles together
// each thread computes one element of the tile of $C$
for (int k = 0; k < BLOCK_SIZE; ++k) {
    Csub += As[ty][k] * Bs[k][tx];
}
// synchronize to make sure that the preceding computation is done before loading two new tiles of $A$ and $B$ in the next iteration
__syncthreads();
Once computed, the working tile of $C$ is written to global memory.

```c
// Write the working tile of $C$ to global memory;
// each thread writes one element
int c = wb * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wb * ty + tx] = Csub;
```
Matrix multiplication (14/16)

- Each thread block should have many threads:
  - TILE_WIDTH = 16 implies 16 × 16 = 256 threads

- There should be many thread blocks:
  - A 1024 × 1024 matrix would require 4096 thread blocks.
  - Since one streaming multiprocessor (SM) can handle 768 threads, each SM will process 3 thread blocks, leading it **full occupancy**.

- Each thread block performs 2 × 256 reads of a 4-byte float while performing 256 × (2 × 16) = 8,192 fp ops:
  - Memory bandwidth is no longer limiting factor
Matrix multiplication (15/16)

- Experimentation performed on a GT200.
- **Tiling** and using *shared memory* were clearly worth the effort.
Matrix multiplication (16/16)

- Effective use of different memory resources reduces the number of accesses to global memory
- But these resources are finite!
- The more memory locations each thread requires, the fewer threads an SM can accommodate.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Per GT200 SM</th>
<th>Full Occupancy on GT200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>16384</td>
<td>&lt;= 16384 / 768 threads = 21 per thread</td>
</tr>
<tr>
<td><strong><strong>shared</strong></strong> Memory</td>
<td>16KB</td>
<td>&lt;= 16KB / 8 blocks = 2KB per block</td>
</tr>
</tbody>
</table>
Plan

1. Data locality and cache misses
   - Hierarchical memories and their impact on our programs
   - Cache complexity and cache-oblivious algorithms put into practice
   - A detailed case study: counting sort

2. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join multithreaded programming model
   - Anticipating parallelization overheads
   - Practical issues and optimization tricks

3. GPU programming
   - The CUDA programming and memory models

4. CUDA Programming: more details and examples

5. CUDA Programming: more details and examples
   - Tiled matrix multiplication in CUDA
   - Optimizing Matrix Transpose with CUDA
   - CUDA programming practices
Matrix transpose characteristics (1/2)

- We optimize a transposition code for a matrix of floats. This operates out-of-place:
  - input and output matrices address separate memory locations.
- For simplicity, we consider an $n \times n$ matrix where 32 divides $n$.
- We focus on the device code:
  - the host code performs typical tasks: data allocation and transfer between host and device, the launching and timing of several kernels, result validation, and the deallocation of host and device memory.
- Benchmarks illustrate this section:
  - we compare our matrix transpose kernels against a matrix copy kernel,
  - for each kernel, we compute the effective bandwidth, calculated in GB/s as twice the size of the matrix (once for reading the matrix and once for writing) divided by the time of execution,
  - Each operation is run NUM_REFS times (for normalizing the measurements),
  - This looping is performed once over the kernel and once within the kernel,
  - The difference between these two timings is kernel launch and synchronization overheads.
Matrix transpose characteristics (2/2)

- We present hereafter different kernels called from the host code, each addressing different performance issues.

- All kernels in this study launch thread blocks of dimension 32x8, where each block transposes (or copies) a tile of dimension 32x32.

- As such, the parameters TILE_DIM and BLOCK_ROWS are set to 32 and 8, respectively.

- Using a thread block with fewer threads than elements in a tile is advantageous for the matrix transpose:
  - each thread transposes several matrix elements, four in our case, and much of the cost of calculating the indices is amortized over these elements.

- This study is based on a technical report by Greg Ruetsch (NVIDIA) and Paulius Micikevicius (NVIDIA).
__global__ void copy(float *odata, float* idata, int width, int height, int nreps)
{
    int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
    int index = xIndex + width*yIndex;

    for (int r=0; r < nreps; r++) { // normalization outer loop
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
            odata[index+i*width] = idata[index+i*width];
        }
    }
}

A simple copy kernel (1/2)
A simple copy kernel (2/2)

- `odata` and `idata` are pointers to the input and output matrices,
- `width` and `height` are the matrix x and y dimensions,
- `nreps` determines how many times the loop over data movement between matrices is performed.
- In this kernel, `xIndex` and `yIndex` are global 2D matrix indices,
  used to calculate `index`, the 1D index used to access matrix elements.

```c
__global__ void copy(float *odata, float* idata, int width, int height, int nreps)
{
    int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
    int index = xIndex + width*yIndex;

    for (int r=0; r < nreps; r++) {
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
            odata[index+i*width] = idata[index+i*width];
        }
    }
}
```
A naive transpose kernel

```c
__global__ void transposeNaive(float *odata, float* idata, 
    int width, int height, int nreps)
{
    int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
    int index_in = xIndex + width * yIndex;
    int index_out = yIndex + height * xIndex;
    for (int r=0; r < nreps; r++) {
        for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
            odata[index_out+i] = idata[index_in+i*width];
        }
    }
}
```
Naive transpose kernel vs copy kernel

The performance of these two kernels on a 2048x2048 matrix using a GTX280 is given in the following table:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>105.14</td>
</tr>
<tr>
<td>naive transpose</td>
<td>18.82</td>
</tr>
</tbody>
</table>

The minor differences in code between the copy and naive transpose kernels have a profound effect on performance.
Coalesced Transpose (1/11)

Because device memory has a much higher latency and lower bandwidth than on-chip memory, special attention must be paid to: how global memory accesses are performed?

The simultaneous global memory accesses by each thread of a half-warp (16 threads on G80) during the execution of a single read or write instruction will be coalesced into a single access if:

1. The size of the memory element accessed by each thread is either 4, 8, or 16 bytes.
2. The address of the first element is aligned to 16 times the element’s size.
3. The elements form a contiguous block of memory.
4. The $i$-th element is accessed by the $i$-th thread in the half-warp.

Last two requirements are relaxed with compute capabilities of 1.2.

Coalescing happens even if some threads do not access memory (divergent warp)
Coalesced Transpose (2/11)

Coalesced float memory access

Coalesced float memory access (divergent warp)
Coalesced Transpose (3/11)
Coalesced Transpose (4/11)
Coalesced Transpose (5/11)

- Allocating device memory through `cudaMalloc()` and choosing `TILE_DIM` to be a multiple of 16 ensures alignment with a segment of memory, therefore all loads from `idata` are coalesced.

- Coalescing behavior differs between the simple copy and naive transpose kernels when writing to `odata`.

- In the case of the naive transpose, for each iteration of the `i`-loop a half warp writes one half of a column of floats to different segments of memory:
  - resulting in 16 separate memory transactions,
  - regardless of the compute capability.
Coalesced Transpose (6/11)

- The way to avoid uncoalesced global memory access is
  1. to read the data into shared memory and,
  2. have each half warp access non-contiguous locations in shared memory in order to write contiguous data to odata.

- There is no performance penalty for non-contiguous access patterns in shared memory as there is in global memory.

- a __syncthreads() call is required to ensure that all reads from idata to shared memory have completed before writes from shared memory to odata commence.
Coalesced Transpose (7/11)

```c
__global__ void transposeCoalesced(float *odata,
    float *idata, int width, int height) // no nreps param
{
    __shared__ float tile[TILE_DIM][TILE_DIM];
    int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
    int index_in = xIndex + (yIndex)*width;
    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
    int index_out = xIndex + (yIndex)*height;
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
        tile[threadIdx.y+i][threadIdx.x] =
            idata[index_in+i*width];
    } __syncthreads();
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
        odata[index_out+i*height] =
            tile[threadIdx.x][threadIdx.y+i];
    }
}
```
The half warp writes four half rows of the idata matrix tile to the shared memory 32x32 array tile indicated by the yellow line segments.

After a `__syncthreads()` call to ensure all writes to tile are completed,

the half warp writes four half columns of tile to four half rows of an odata matrix tile, indicated by the green line segments.
Coalesced Transpose (9/11)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>105.14</td>
</tr>
<tr>
<td>shared memory copy</td>
<td>104.49</td>
</tr>
<tr>
<td>naive transpose</td>
<td>18.82</td>
</tr>
</tbody>
</table>

While there is a dramatic increase in effective bandwidth of the coalesced transpose over the naive transpose, there still remains a large performance gap between the coalesced transpose and the copy:

- One possible cause of this performance gap could be the synchronization barrier required in the coalesced transpose.
- This can be easily assessed using the following copy kernel which utilizes shared memory and contains a `__syncthreads()` call.
Coalesced Transpose (10/11)

```c
__global__ void copySharedMem(float *odata, float *idata,
                            int width, int height) // no nreps param
{
    __shared__ float tile[TILE_DIM][TILE_DIM];
    int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
    int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
    int index = xIndex + width*yIndex;
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
        tile[threadIdx.y+i][threadIdx.x] =
            idata[index+i*width];
    }
    __syncthreads();
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
        odata[index+i*width] =
            tile[threadIdx.y+i][threadIdx.x];
    }
}
```
Coalesced Transpose (11/11)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>105.14</td>
</tr>
<tr>
<td>shared memory copy</td>
<td>104.49</td>
</tr>
<tr>
<td>naive transpose</td>
<td>18.82</td>
</tr>
<tr>
<td>coalesced transpose</td>
<td>51.42</td>
</tr>
</tbody>
</table>

The shared memory copy results seem to suggest that the use of shared memory with a synchronization barrier has little effect on the performance, certainly as far as the *Loop in kernel* column indicates when comparing the simple copy and shared memory copy.
Shared memory bank conflicts (1/6)

1. Shared memory is divided into 16 equally-sized memory modules, called **banks**, which are organized such that successive 32-bit words are assigned to successive banks.

2. These banks can be accessed simultaneously, and to achieve maximum bandwidth to and from shared memory the **threads in a half warp should access shared memory associated with different banks**.

3. The **exception to this rule** is when all threads in a half warp read the same shared memory address, which results in a broadcast where the data at that address is sent to all threads of the half warp in one transaction.

4. One can use the **warp_serialize** flag when profiling CUDA applications to determine whether shared memory bank conflicts occur in any kernel.
Shared memory bank conflicts (2/6)
Shared memory bank conflicts (3/6)
The coalesced transpose uses a $32 \times 32$ shared memory array of floats.

For this sized array, all data in columns $k$ and $k+16$ are mapped to the same bank.

As a result, when writing partial columns from tile in shared memory to rows in odata the half warp experiences a 16-way bank conflict and serializes the request.

A simple way to avoid this conflict is to pad the shared memory array by one column:

```c
__shared__ float tile[TILE_DIM][TILE_DIM+1];
```
The padding does not affect shared memory bank access pattern when writing a half warp to shared memory, which remains conflict free,

but by adding a single column now the access of a half warp of data in a column is also conflict free.

The performance of the kernel, now coalesced and memory bank conflict free, is added to our table on the next slide.
While padding the shared memory array did eliminate shared memory bank conflicts, as was confirmed by checking the warp_serialization flag with the CUDA profiler, it has little effect (when implemented at this stage) on performance.

As a result, there is still a large performance gap between the coalesced and shared memory bank conflict-free transpose and the shared memory copy.
Plan

1. Data locality and cache misses
   - Hierarchical memories and their impact on our programs
   - Cache complexity and cache-oblivious algorithms put into practice
   - A detailed case study: counting sort

2. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join multithreaded programming model
   - Anticipating parallelization overheads
   - Practical issues and optimization tricks

3. GPU programming
   - The CUDA programming and memory models

4. CUDA Programming: more details and examples

5. CUDA Programming: more details and examples
   - Tiled matrix multiplication in CUDA
   - Optimizing Matrix Transpose with CUDA
   - CUDA programming practices
Four principles

- Expose as much parallelism as possible
- Optimize memory usage for maximum bandwidth
- Maximize occupancy to hide latency
- Optimize instruction usage for maximum throughput
Exposé Parallelism

- Structure algorithm to maximize independent parallelism
- If threads of same block need to communicate, use shared memory and __syncthreads()
- If threads of different blocks need to communicate, use global memory and split computation into multiple kernels
- Recall that there is no synchronization mechanism between blocks
- High parallelism is especially important to hide memory latency by overlapping memory accesses with computation
- Take advantage of asynchronous kernel launches by overlapping CPU computations with kernel execution.
Optimize Memory Usage: Basic Strategies

- Processing data is cheaper than moving it around:
  - Especially for GPUs as they devote many more transistors to ALUs than memory

- Basic strategies:
  - Maximize use of low-latency, high-bandwidth memory
  - Optimize memory access patterns to maximize bandwidth
  - Leverage parallelism to hide memory latency by overlapping memory accesses with computation as much as possible
  - Write kernels with high arithmetic intensity (ratio of arithmetic operations to memory transactions)
  - Sometimes recompute data rather than cache it
Minimize CPU \(\leftarrow\) GPU Data Transfers

- CPU \(\leftarrow\) GPU memory bandwidth much lower than GPU memory bandwidth

- Minimize CPU \(\leftarrow\) GPU data transfers by moving more code from CPU to GPU
  - Even if sometimes that means running kernels with low parallelism computations
  - Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to CPU memory

- Group data transfers: One large transfer much better than many small ones.
Optimize Memory Access Patterns

- Effective bandwidth can vary by an order of magnitude depending on access pattern:
  - Global memory is not cached on G8x.
  - Global memory has High latency instructions: 400-600 clock cycles
  - Shared memory has low latency: a few clock cycles

- Optimize access patterns to get:
  - Coalesced global memory accesses
  - Shared memory accesses with no or few bank conflicts and
  - to avoid partition camping.
A Common Programming Strategy

1. Partition data into subsets that fit into shared memory
2. Handle each data subset with one thread block
3. Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.
4. Perform the computation on the subset from shared memory.
5. Copy the result from shared memory back to global memory.
A Common Programming Strategy

- Carefully partition data according to access patterns
- If read only, use `__constant__` memory (fast)
- For read/write access within a tile, use `__shared__` memory (fast)
- For read/write scalar access within a thread, use registers (fast)
- R/W inputs/results `cudaMalloc`ed, use global memory (slow)
A Common Programming Strategy

Partition data into subsets that fit into shared memory
A Common Programming Strategy

Handle each data subset with one thread block
A Common Programming Strategy

Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.
A Common Programming Strategy

Perform the computation on the subset from shared memory.
A Common Programming Strategy

Copy the result from shared memory back to global memory.