Optimizing Algorithms and Code for Data Locality and Parallelism

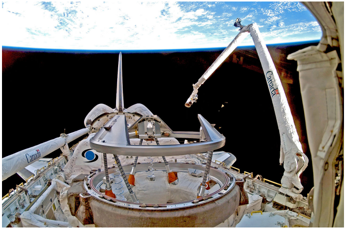
Marc Moreno Maza & Yuzhen Xie

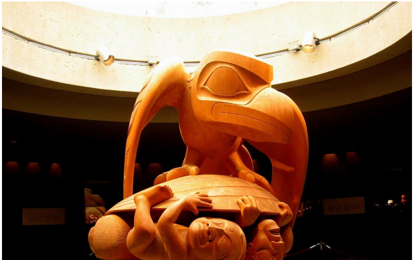
University of Western Ontario, Canada

American University of Beirut
June 16-18, 2014

What is unique to Lebanon and Canada?

Canada in 4 pictures





High-performance computing and symbolic computation

Basic Polynomial Algebra Subprograms

www.bpaslib.org

MetaFork

www.metafork.org

CUDA MODular Polynomial

www.cumodp.org

www.regularchains.org

What is this tutorial 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 tutorial 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 tutorial 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

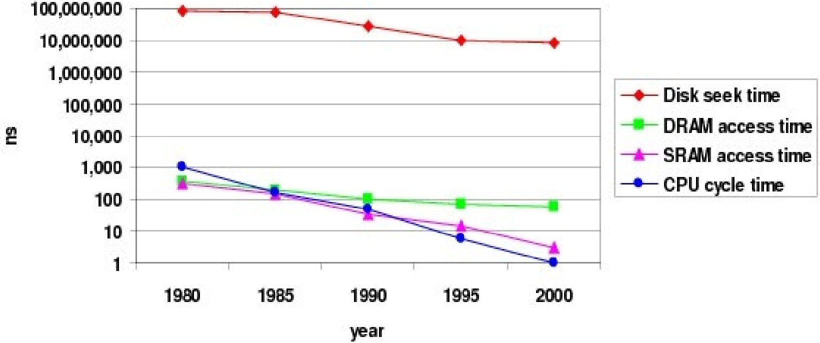
- Computer cache memories have led to introduce a new complexity measure for algorithms and new performance counters for code.
- Optimizing for data locality brings large speedup factors.

Optimizing for parallelism

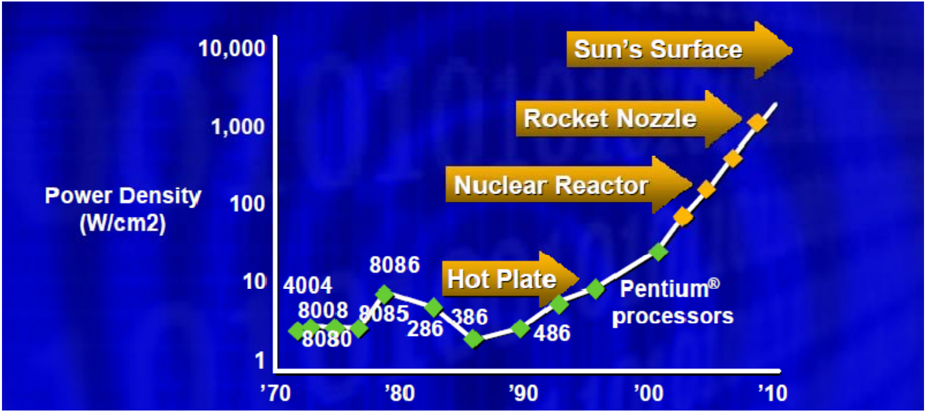
- All recent home and office desktops/laptops are parallel machines; moreover "GPU cards bring supercomputing to the masses."
- Optimizing for parallelism improves the use of computing resources.
- And optimizing for data locality is often a first step!

The CPU-Memory Gap

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



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 tutorial?

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 and references

Acknowledgments.

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

References.

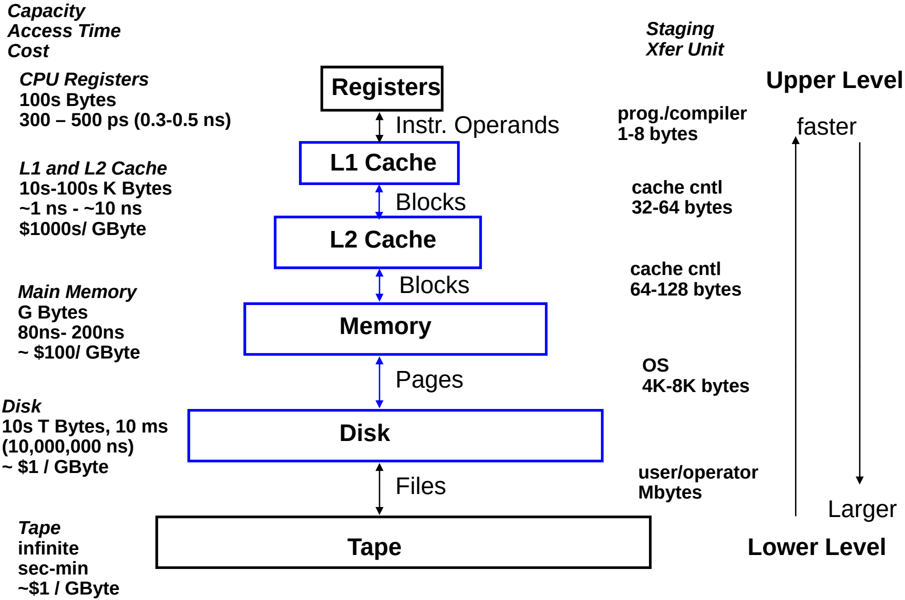
- *The Implementation of the Cilk-5 Multithreaded Language* by Matteo Frigo Charles E. Leiserson Keith H. Randall.
- *Cache-Oblivious Algorithms* by Matteo Frigo, Charles E. Leiserson, Harald Prokop and Sridhar Ramachandran.
- *The Cache Complexity of Multithreaded Cache Oblivious Algorithms* by Matteo Frigo and Volker Strumpen.
- *How To Write Fast Numerical Code: A Small Introduction* by Srinivas Chellappa, Franz Franchetti, and Markus Pueschel.
- *Models of Computation: Exploring the Power of Computing* by John E. Savage.
- `http://developer.nvidia.com/category/zone/cuda-zone`
- `http://www.csd.uwo.ca/~moreno/HPC-Resources.html`

Plan

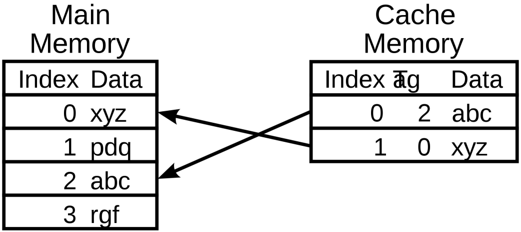
1. Data locality and cache misses
   - Hierarchical memories and their impact on our programs
   - Cache complexity and cache-oblivious algorithms put into practice
   - A detailed case study: counting sort
2. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join multithreaded programming model
   - Anticipating parallelization overheads
3. GPU programming
   - The CUDA programming and memory models
   - 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
3. GPU programming
   - The CUDA programming and memory models
   - Tiled matrix multiplication in CUDA
   - Optimizing Matrix Transpose with CUDA
   - CUDA programming practices

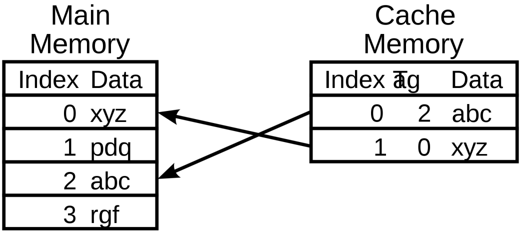

Capacity
Access Time
Cost
Staging
Xfer Unit
Upper Level
CPU Registers
100s Bytes
300 – 500 ps (0.3-0.5 ns)
Registers
Instr. Operands
prog./compiler
1-8 bytes
faster
L1 and L2 Cache
10s-100s K Bytes
~1 ns - ~10 ns
$1000s/ GByte
L1 Cache
Blocks
cache cntl
32-64 bytes
L2 Cache
Blocks
cache cntl
64-128 bytes
Main Memory
G Bytes
80ns- 200ns
~ $100/ GByte
Memory
Pages
OS
4K-8K bytes
Disk
10s T Bytes, 10 ms
(10,000,000 ns)
~ $1 / GByte
Disk
Files
user/operator
Mbytes
Larger
Tape
infinite
sec-min
~$1 / GByte
Tape
Lower Level

CPU Cache (1/7)



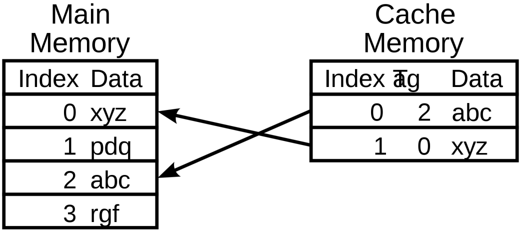
- A CPU cache is an auxiliary memory which is smaller, faster memory than the main memory and which stores copies of the main memory locations that are expectedly frequently used.
- Most modern desktop and server CPUs have at least three independent caches: the data cache, the instruction cache and the translation look-aside buffer.

CPU Cache (2/7)



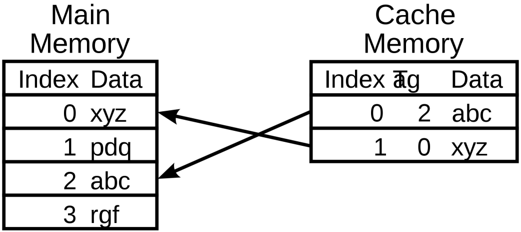
- Each location in each memory (main or cache) has
  - a datum (cache line) which ranges between 8 and 512 bytes in size, while a datum requested by a CPU instruction ranges between 1 and 16.
  - a unique index (called address in the case of the main memory)
- In the cache, each location has also a tag (storing the address of the corresponding cached datum).

CPU Cache (3/7)



- 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.

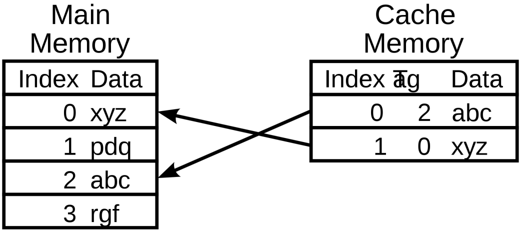
CPU Cache (4/7)



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

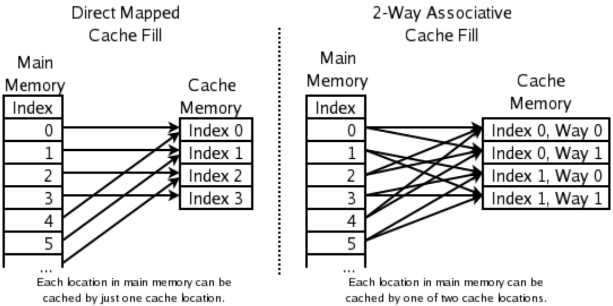
- out-of-order execution: attempt to execute independent instructions arising after the instruction that is waiting due to the cache miss
- hyper-threading (HT): allows an alternate thread to use the CPU

CPU Cache (5/7)



- 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 (6/7)



- 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

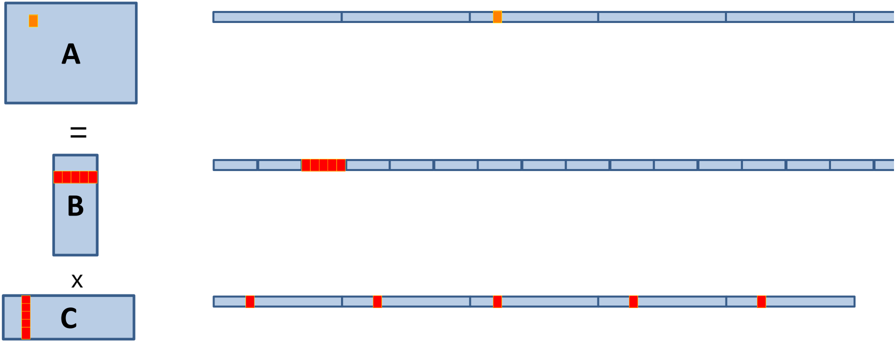
Cache issues

- **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

```
#define IND(A, x, y, d) A[(x)*(d)+(y)]
uint64_t testMM(const int x, const int y, const int z)
{
  double *A;  *B; *C;
        long started, ended;
        float timeTaken;
        int i, j, k;
        srand(getSeed());
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i++)
          for (j = 0; j < y; j++)
             for (k = 0; k < z; k++)
                    //  A[i][j] += B[i][k] + C[k][j];
                    IND(A,i,j,y) += IND(B,i,k,z) * IND(C,k,j,y);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
  return timeTaken;
}
```

Issues with matrix representation

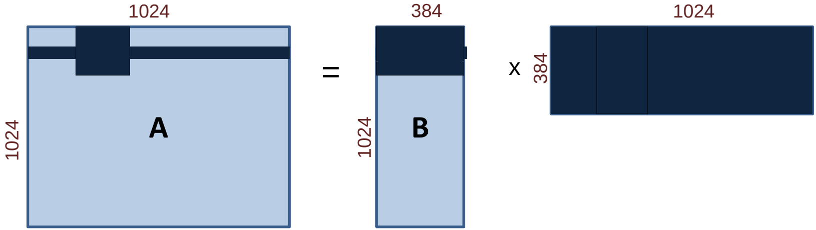


- Contiguous accesses are better:
  - Data fetch as cache line (Core 2 Duo 64 byte per cache line)
  - With contiguous data, a single cache fetch supports 8 reads of doubles.
  - Transposing the matrix `C` should reduce L1 cache misses!

Transposing for optimizing spatial locality

```
float testMM(const int x, const int y, const int z)
{
  double *A; double *B; double *C; double *Cx;
        long started, ended; float timeTaken; int i, j, k;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        Cx = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for(j =0; j < y; j++)
          for(k=0; k < z; k++)
            IND(Cx,j,k,z) = IND(C,k,j,y);
        for (i = 0; i < x; i++)
          for (j = 0; j < y; j++)
             for (k = 0; k < z; k++)
               IND(A, i, j, y) += IND(B, i, k, z)*IND(Cx, j, k, z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
  return timeTaken;
}
```

Issues with data reuse



- Naive calculation of a row of A, so computing 1024 coefficients: 1024 accesses in A, 384 in B and $1024 \times 384 = 393,216$ in C. Total $= 394,524$.
- Computing a 32×32-block of A, so computing again 1024 coefficients: 1024 accesses in A, 384×32 in B and 32×384 in C. Total $= 25,600$.
- The iteration space is traversed so as to reduce memory accesses.

Blocking for optimizing temporal locality

```
float testMM(const int x, const int y, const int z)
{
        double *A; double *B; double *C;
        long started, ended; float timeTaken; int i, j, k, i0, j0, k0;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i += BLOCK_X)
          for (j = 0; j < y; j += BLOCK_Y)
            for (k = 0; k < z; k += BLOCK_Z)
              for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
                for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
                   for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
                       IND(A,i0,j0,y) += IND(B,i0,k0,z) * IND(C,k0,j0,y);
         ended = example_get_time();
         timeTaken = (ended - started)/1.f;
   return timeTaken;
}
```

Transposing and blocking for optimizing data locality

```
float testMM(const int x, const int y, const int z)
{
        double *A; double *B; double *C, 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;
```

Experimental results

Computing the product of two $n \times n$ matrices on my laptop (Quad-core Intel i7-3630QM CPU @ 2.40GHz L2 cache 6144 KB, 8 GBytes of RAM)

n	naive	transposed	8×8-tiled	t. & t.
1024	7854	1086	1105	999
2048	8335	8646	10166	7990
4096	747100	69149	100538	69745
8192	6914349	546585	823525	562433

Timings are in milliseconds.

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

Experimental results: going further . . .

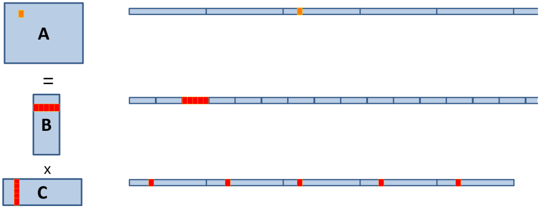
Other performance counters

Hardware count events

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

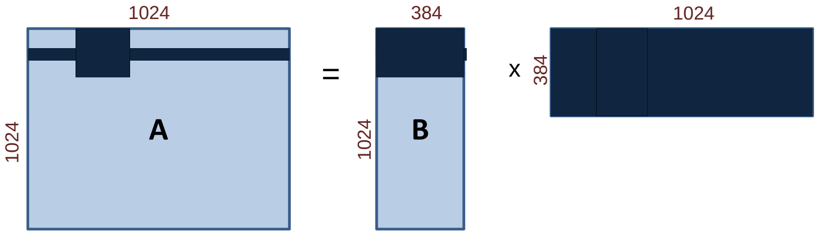
	CPI		L1 Miss Rate		L2 Miss Rate	Percent SSE Instructions	Instructions Retired	
In C	4.78		0.24		0.02	43%	13,137,280,000	
		5x		2x				1x
Transposed	1.13		0.15		0.02	50%	13,001,486,336	
		3x		8x				0.8x
Tiled	0.49		0.02		0	39%	18,044,811,264	

Analyzing cache misses in the naive and transposed multiplication



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

Analyzing cache misses in the tiled multiplication



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

Plan

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
3. GPU programming
   - The CUDA programming and memory models
   - Tiled matrix multiplication in CUDA
   - Optimizing Matrix Transpose with CUDA
   - CUDA programming practices

The (Z, L) ideal cache model (1/4)

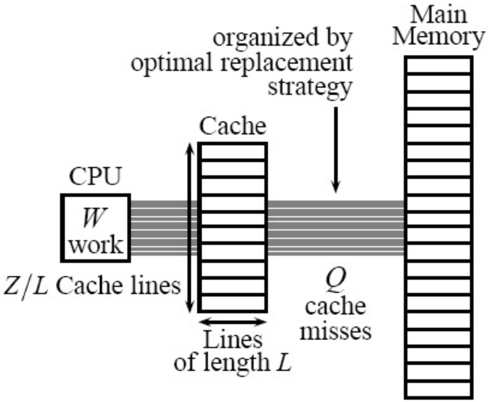


Figure 1: The ideal-cache model

The (Z, L) ideal cache model (2/4)

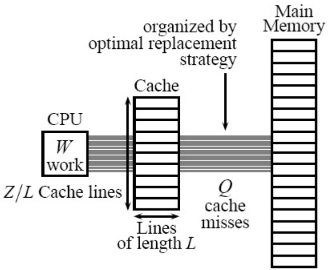


Figure 1: The ideal-cache model

- 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)

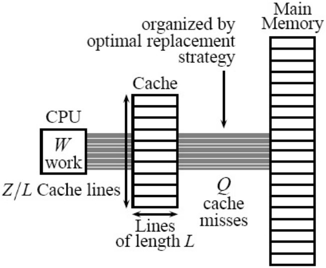


Figure 1: The ideal-cache model

- 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.

The (Z, L) ideal cache model (4/4)

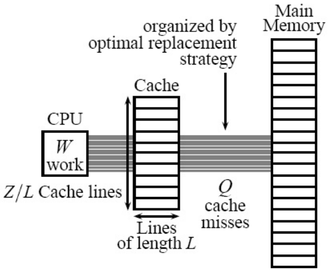


Figure 1: The ideal-cache model

- 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.

Cache complexity

- For an algorithm with an input of size n, he 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

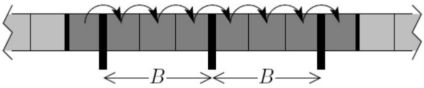


Figure 2. Scanning an array of N elements arbitrarily aligned with blocks may cost one more memory transfer than $\lceil N/B \rceil$.

- 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.

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)

```
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)

```
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 Rec-Transpose below incurs $\Theta(1 + mn/L)$ cache misses, which is optimal.
  1. If $n \geq m$, the Rec-Transpose algorithm partitions
  $$A = (A_1 \ A_2) \ , \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$
  and recursively executes Rec-Transpose(A_1, B_1) and Rec-Transpose(A_2, B_2).
  2. If $m > n$, the Rec-Transpose algorithm partitions
  $$A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \ , \quad B = (B_1 \ B_2)$$
  and recursively executes Rec-Transpose(A_1, B_1) and Rec-Transpose(A_2, B_2).

Cache-oblivious matrix transposition works in practice!

size	Naive	Cache-oblivious	ratio
5000x5000	126	79	1.59
10000x10000	627	311	2.02
20000x20000	4373	1244	3.52
30000x30000	23603	2734	8.63
40000x40000	62432	4963	12.58

- `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:

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}, \qquad (1)$$

$$\begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 , \qquad (2)$$

$$A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} . \qquad (3)$$

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

```
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
```

Counting sort: cache complexity analysis (1/9)

```
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. . . . to compute k.

Counting sort: cache complexity analysis (2/9)

```
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.

Counting sort: cache complexity analysis (3/9)

```
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. ...cache misses to initialize Count.

Counting sort: cache complexity analysis (4/9)

```
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 (5/9)

```
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 (6/9)

```
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).

Counting sort: cache complexity analysis (7/9)

```
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. ... cache misses for the prefix sum.

Counting sort: cache complexity analysis (8/9)

```
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.

Counting sort: cache complexity analysis (9/9)

```
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 (10/9)

```
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. $n/L + n + n$ cache misses for building Output (worst case).

Counting sort: cache complexity analysis (11/9)

```
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. $n/L + n + n$ cache misses for building Output (worst case).

   Total: $3n + 3n/L + 2k/L$ cache misses (worst case).

Counting sort: cache complexity analysis: explanations

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

Total: $3n + 3n/L + 2k/L$ 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

```
alloacate 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
alloacate 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, $m + mL \leq Z$.

Counting sort: cache complexity with bucketing

```
alloacate 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
alloacate 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 $i = 0 \cdots m-1$, $h + |\{\texttt{key} \in [ih, (i+1)h-1]\}| < Z$ and $m < Z/(1+L)$. These two are very realistic assumption considering today's cache size.
- And the total complexity becomes;

$$
\begin{array}{rcl}
Q_{\rm total} & = & Q_{\rm preprocessing} + Q_{\rm sorting} \\
 & = & Q_{\rm preprocessing} + m\, Q_{\rm sortingofonebucket} \\
 & = & Q_{\rm preprocessing} + m\, (3\frac{n}{m\,\mathbf{L}} + 3\frac{n}{m} + 2\frac{k}{mL}) \\
 & = & Q_{\rm preprocessing} + 6n/L + 2k/L \\
 & = & 3n/L + 4m/L + m + 6n/L + 2k/L \\
 & = & 9n/L + 4m/L + m + 2k/L
\end{array}
$$

Instead of $3n$+$3n/L + 2k/L$ for the naive counting sort.

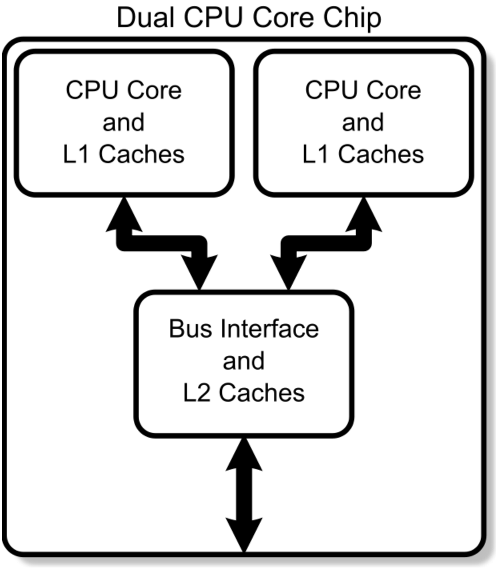
Cache friendly counting sort: experimental results

- 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 $[0, n]$.

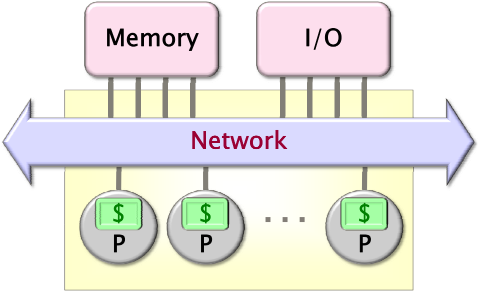
n	classical counting sort	cache-oblivious counting sort (preprocessing + sorting)
100000000	13.74	4.66 (3.04 + 1.62)
200000000	30.20	9.93 (6.16 + 3.77)
300000000	50.19	16.02 (9.32 + 6.70)
400000000	71.55	22.13 (12.50 +9.63)
500000000	94.32	28.37 (15.71 + 12.66)
600000000	116.74	34.61 (18.95 + 15.66)

Summary and notes

Plan



- A multi-core processor is an integrated circuit to which two or more individual processors (called cores in this sense) have been attached.



Chip Multiprocessor (CMP)

- 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)

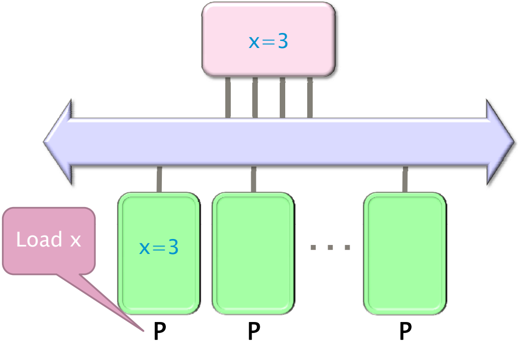


Figure: Processor P_1 reads x=3 first from the backing store (higher-level memory)

Cache Coherence (2/6)

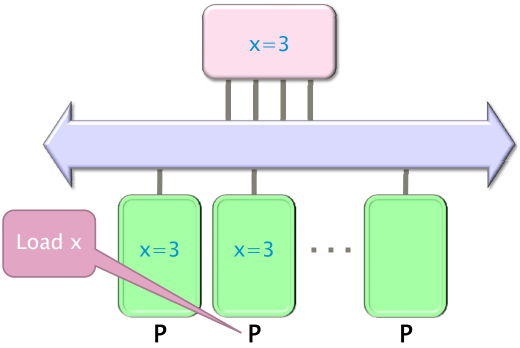


Figure: Next, Processor P_2 loads x=3 from the same memory

Cache Coherence (3/6)

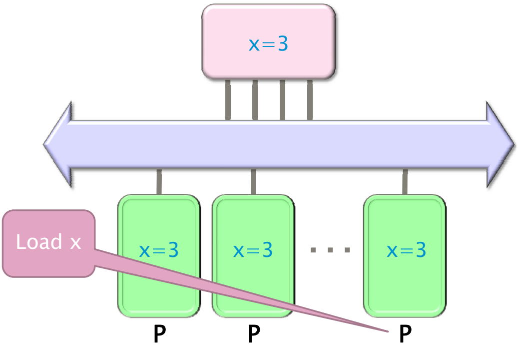


Figure: Processor P_4 loads x=3 from the same memory

Cache Coherence (4/6)

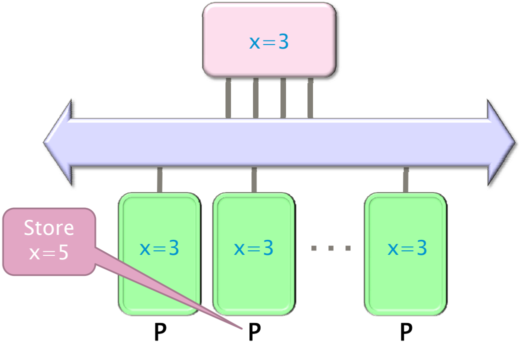


Figure: Processor P_2 issues a write x=5

Cache Coherence (5/6)

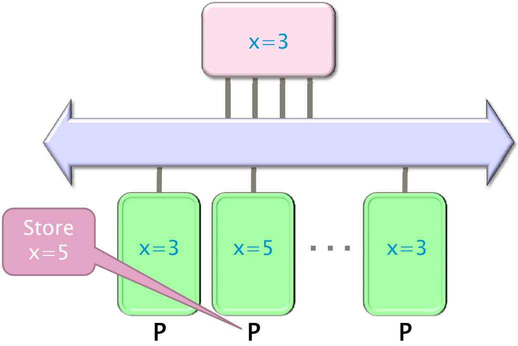


Figure: Processor P_2 writes x=5 in his local cache

Cache Coherence (6/6)

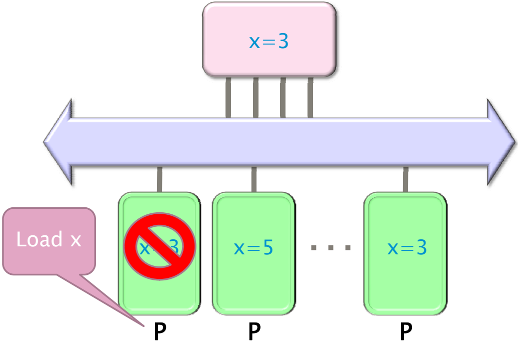


Figure: Processor P_1 issues a read x, which is now invalid in its cache

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

```
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

$$\underset{A}{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}} \Rightarrow \underset{A^T}{\begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix}}$$

```
// indices run from 0, not 1
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 may execute in parallel.

Serial Semantics (1/2)

- 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

```
#define cilk_for for
#define cilk_spawn
#define cilk_sync
```

Serial Semantics (2/2)

```
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);
  }
}
```

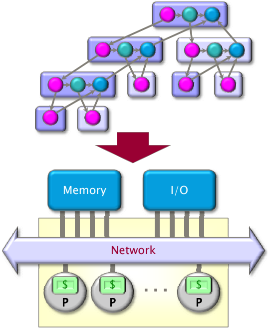
Cilk++ source

↓

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

Serialization

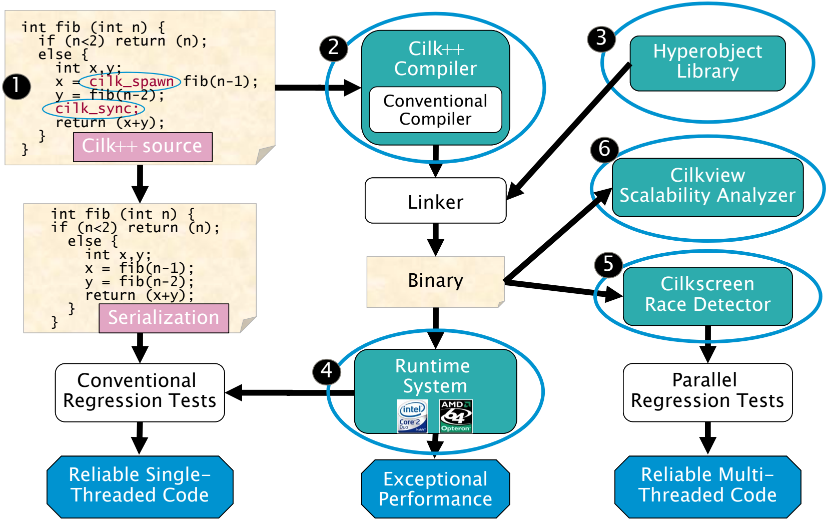
Scheduling



A **scheduler**'s job is to map a computation to particular processors. Such a mapping is called a **schedule**.

- If decisions are made at runtime, the scheduler is *online*, otherwise, it is *offline*
- `CilkPlus`'s scheduler maps strands onto processors dynamically at runtime.

The CilkPlus Platform



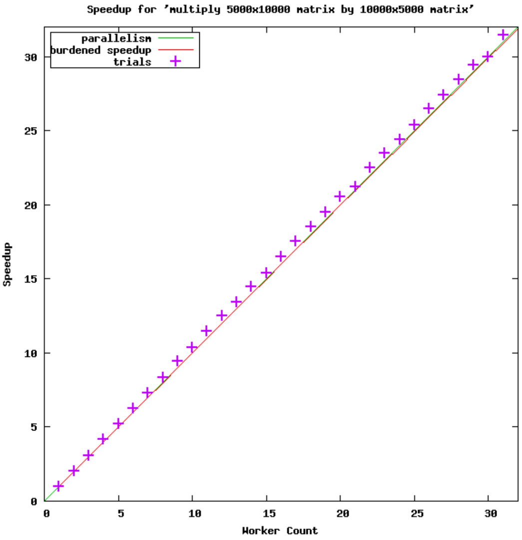
Benchmarks for the parallel version of the cache-oblivious mm

Multiplying a 4000x8000 matrix by a 8000x4000 matrix

- on 32 cores = 8 sockets x 4 cores (Quad Core AMD Opteron 8354) per socket.
- The 32 cores share a L3 32-way set-associative cache of 2 Mbytes.

#core	Elision (s)	Parallel (s)	speedup
8	420.906	51.365	8.19
16	432.419	25.845	16.73
24	413.681	17.361	23.83
32	389.300	13.051	29.83

So does the (tuned) cache-oblivious matrix multiplication



Plan

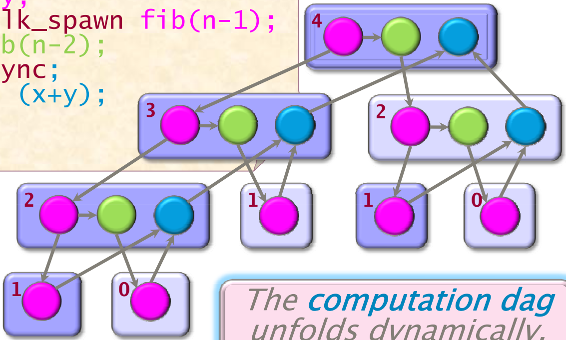
1. Data locality and cache misses
   - Hierarchical memories and their impact on our programs
   - Cache complexity and cache-oblivious algorithms put into practice
   - A detailed case study: counting sort
2. Multicore programming
   - Multicore architectures
   - Cilk / Cilk++ / Cilk Plus
   - The fork-join multithreaded programming model
   - Anticipating parallelization overheads
3. GPU programming
   - The CUDA programming and memory models
   - 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

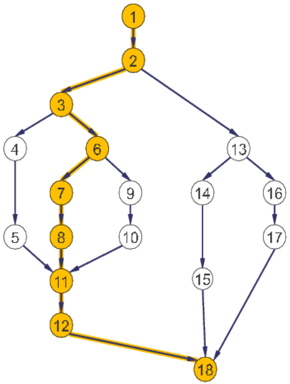


Figure: Instruction stream DAG.

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_∞ 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_∞ : **Parallelism**.

- *Work law:* $T_p \geq T_1/p$.
- *Span law:* $T_p \geq T_\infty$.

For loop parallelism in Cilk++

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}}_{A} \Rightarrow \underbrace{\begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix}}_{A^T}$$

```
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**:

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

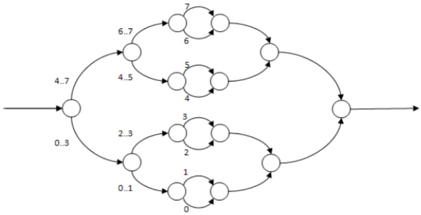
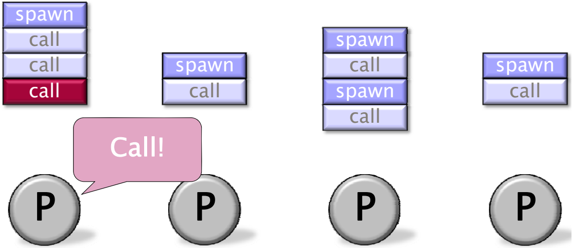
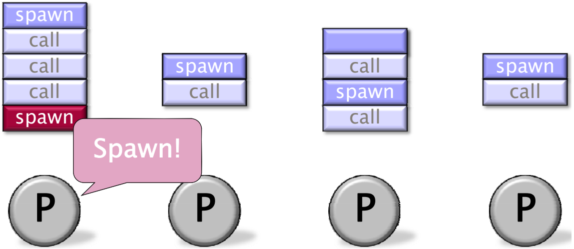


Figure: DAG for a *cilk_for* with 8 iterations.

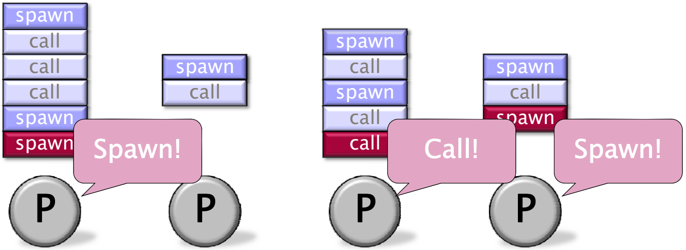
The work-stealing scheduler (1/11)



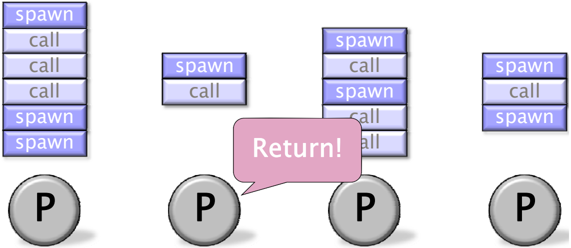
The work-stealing scheduler (2/11)



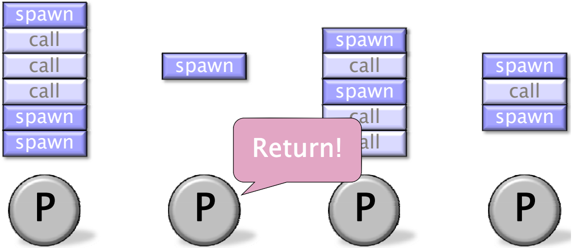
The work-stealing scheduler (3/11)



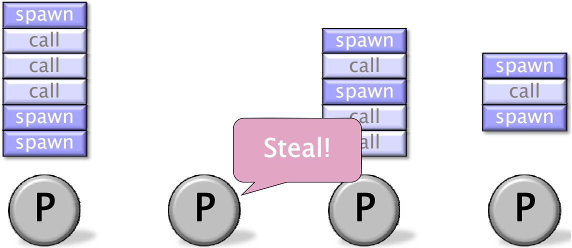
The work-stealing scheduler (4/11)



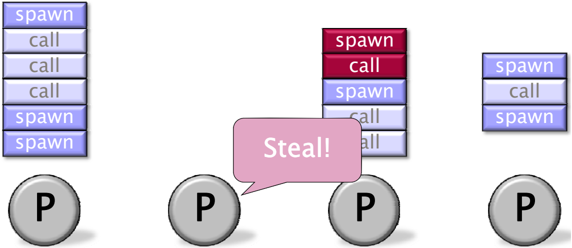
The work-stealing scheduler (5/11)



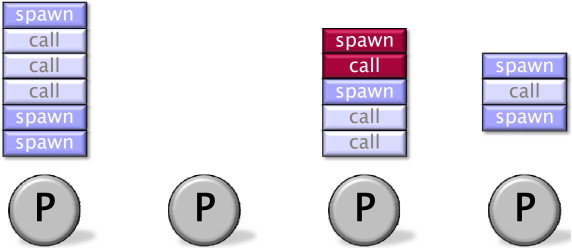
The work-stealing scheduler (6/11)



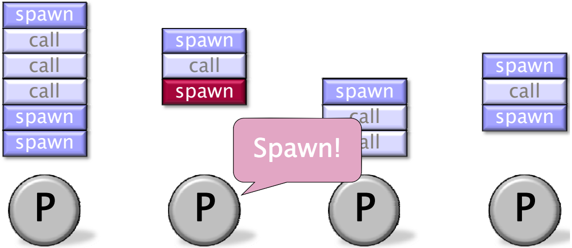
The work-stealing scheduler (7/11)



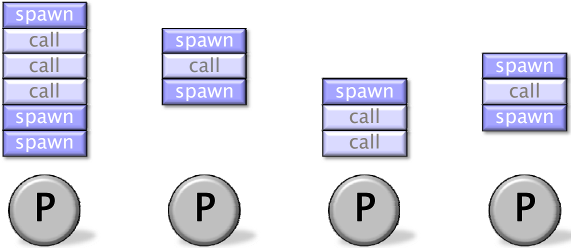
The work-stealing scheduler (8/11)



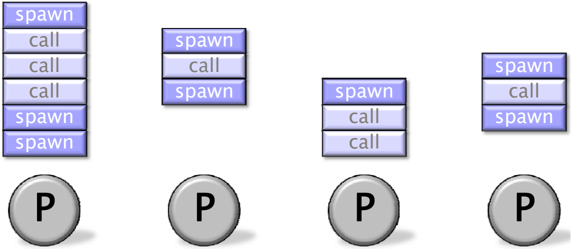
The work-stealing scheduler (9/11)



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.

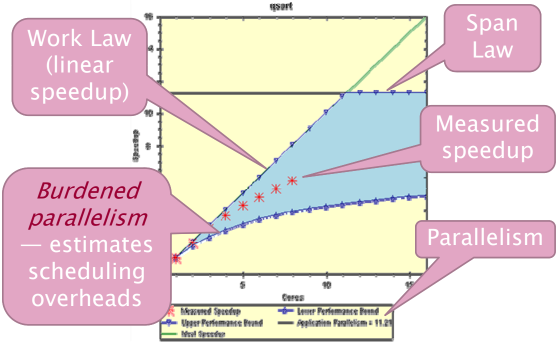
Then, the randomized work-stealing scheduler is expected to run in

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

Overheads and burden

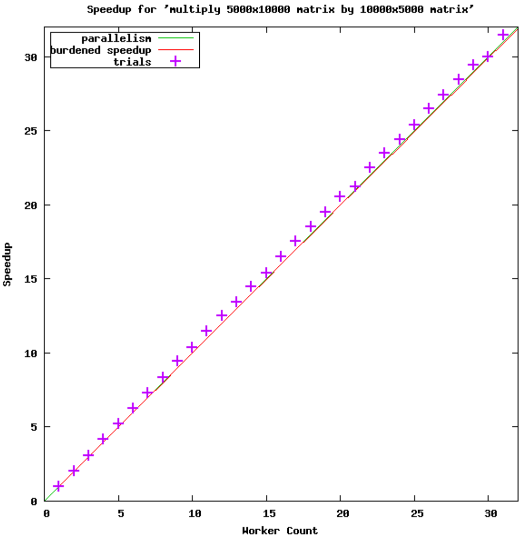
- Many factors (simplification assumptions of the fork-join parallelism model, architecture limitation, costs of executing the parallel constructs, overheads of scheduling) will make T_p larger in practice than $T_1/p + T_\infty$.
- `Cilk++` estimates T_p as $T_p = T_1/p + 1.7 \text{ burden\_span}$, where burden_span is 15000 instructions times the number of continuation edges along the critical path.

Cilkview



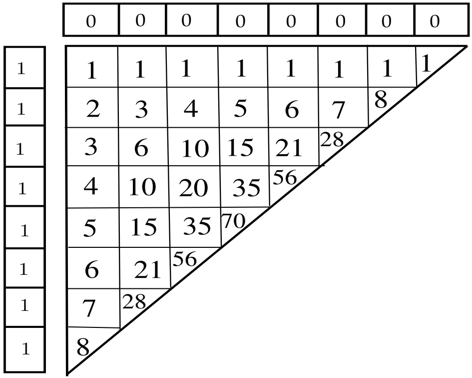
- Cilkview computes work and span to derive upper bounds on parallel performance
- Cilkview also estimates scheduling overhead to compute a burdened span for lower bounds.

Tuned cache-oblivious parallel matrix multiplication



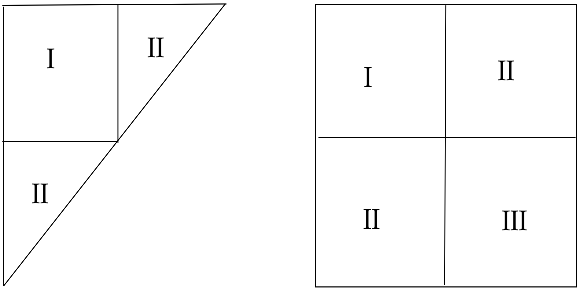
Plan

Pascal Triangle



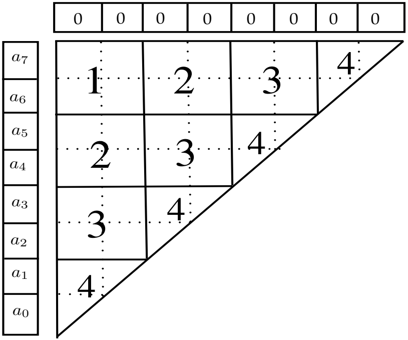
Construction of the Pascal Triangle: nearly the simplest stencil computation!

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.

Blocking strategy: principle



- 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 tress $T_0, T_1, \ldots, T_{n/B-1}$ such that

- T_i is the instruction stream DAG of the `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

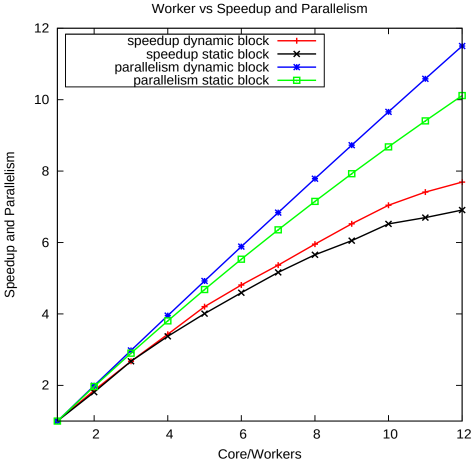
$$S_b(n) = \sum_{i=1}^{n/B} \log(i) = \log(\prod_{i=1}^{n/B} i) = \log(\Gamma(\frac{n}{B} + 1)).$$

Using Stirling's Formula, we deduce

$$S_b(n) \in \Theta\left(\frac{n}{B}\log(\frac{n}{B})\right). \quad (4)$$

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

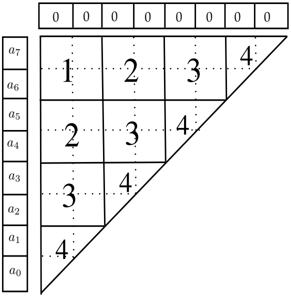


Summary and notes

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



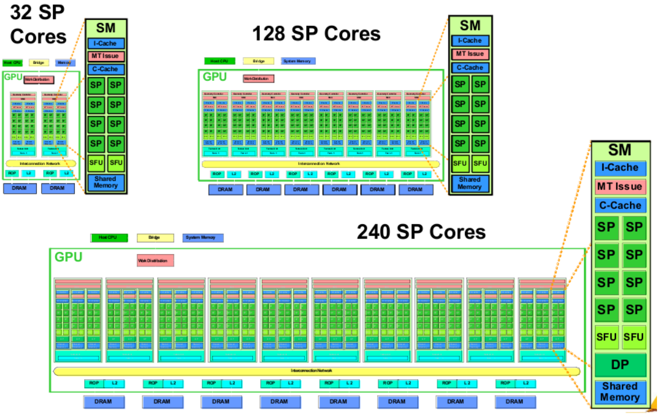
- Consider executing one band after another, where for each band all $B \times B$ blocks are executed concurrently.
- The non-burdened span is in $\Theta(B^2 n/B) = \Theta(n/B)$.
- While the burdened span is

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

Plan

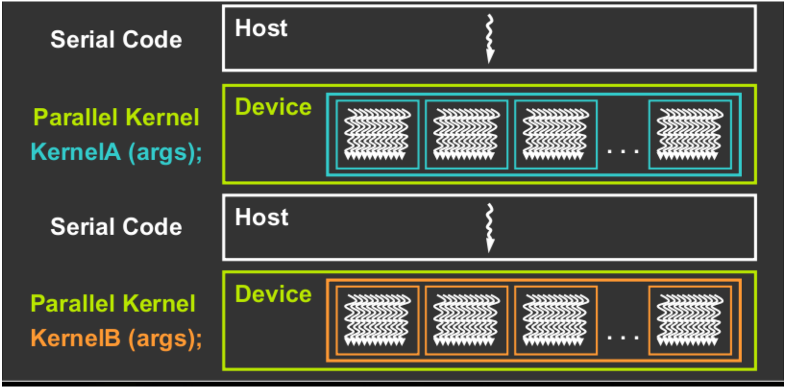
CUDA design goals

- 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



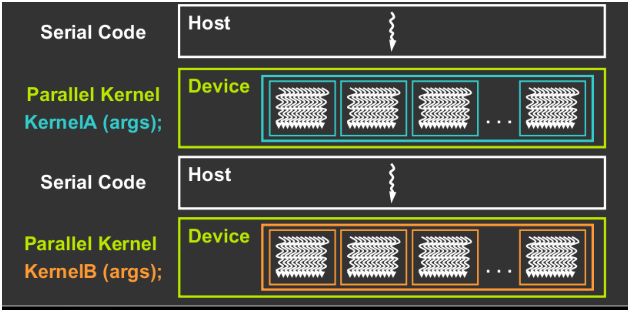
Heterogeneous programming (1/3)

- 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).



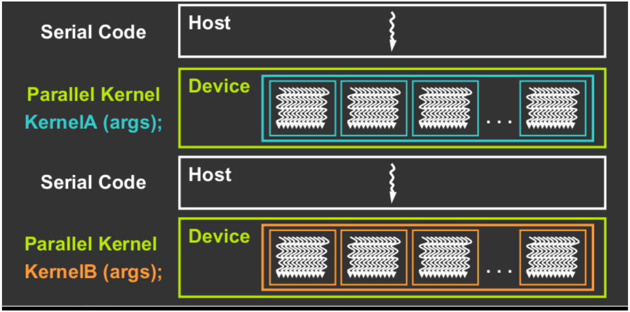
Heterogeneous programming (2/3)

- 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, blockDim=4 -> 4 blocks

```
int idx = blockDim.x * blockId.x + threadIdx.x;
```

blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,15

See our example number 4 in `/usr/local/cs4402/examples/4`

Example: increment array elements (2/2)

CPU program

```
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);
}
```

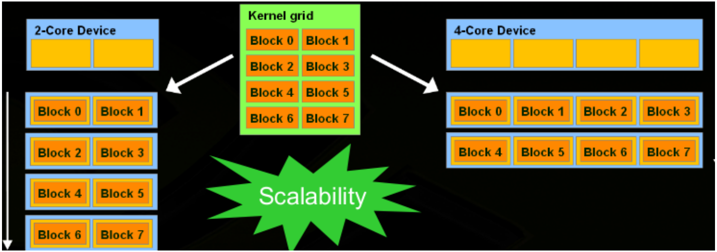
CUDA program

```
__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);
}
```

Thread blocks (1/2)

- 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



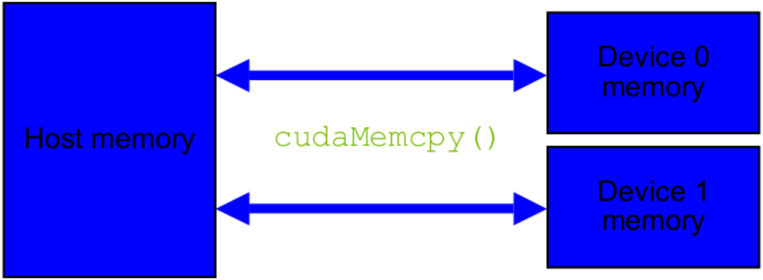
Thread blocks (2/2)

- 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.

Memory hierarchy (1/3)

Host (CPU) memory:

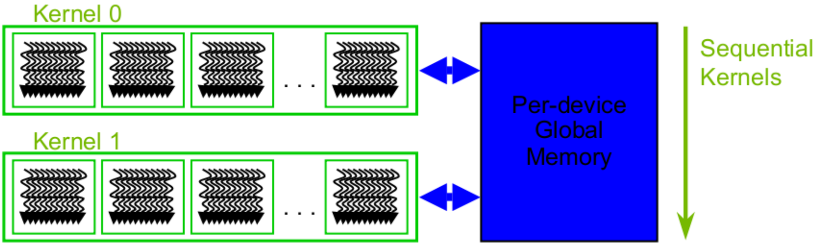
- Not directly accessible by CUDA threads



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



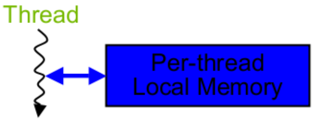
Memory hierarchy (3/3)

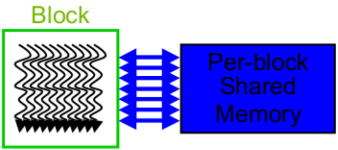
Shared memory:

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

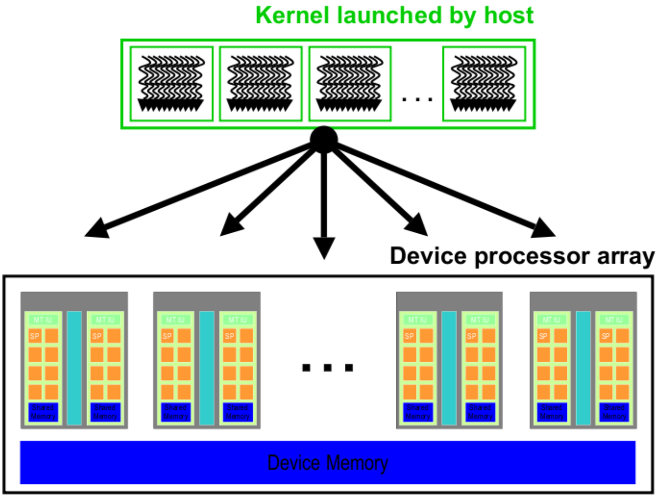
Local storage:

- Each thread has its own local storage
- Data lifetime = thread lifetime

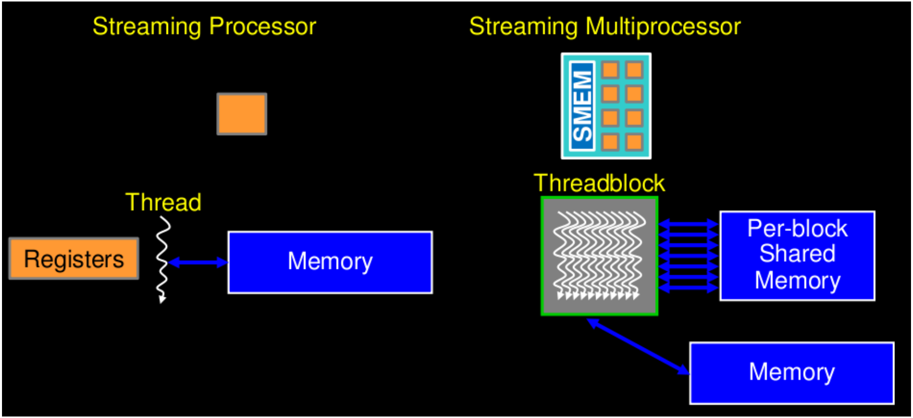




Blocks run on multiprocessors

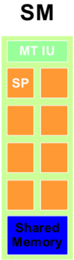


Streaming processors and multiprocessors



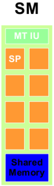
Hardware multithreading

- **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.



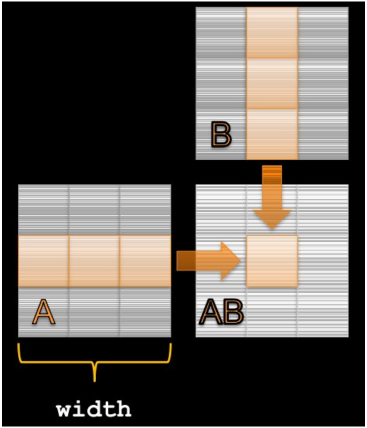
Plan

Matrix multiplication (1/16)

- 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)

```
__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
  - $2\,m\,n\,p$ **reads in total for** $2\,m\,n\,p$ **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:
  - $(m/t)(t\,n)(p/t)$ slots are read in A
  - $(p/t)(t\,n)(m/t)$ slots are read in B, thus
  - $2m\,n\,p/t$ **reads in total for** $2\,m\,n\,p$ **flops.**
- For a CUDA implementation, $t = 16$ such that each tile is computed by one thread block.

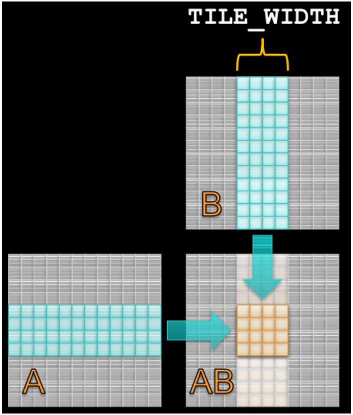
Matrix multiplication (5/16)

- 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 $\mathrm{GMAC} \times \mathrm{Peak\ FLOPS} =$ 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:

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

- Therefore we fix ℓ and then compute $\Sigma_{k=\ell t}^{(\ell+1)t-1} a_{i,k} b_{k,j}$ 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.

```
#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`.

```
int aBegin = wa * BLOCK_SIZE * by;

int aEnd = aBegin + wa - 1;

int aStep = BLOCK_SIZE;
```

Matrix multiplication (10/16)

- Similarly for B we have `bBegin` and `bStep`.
- We will not need a `bEnd` 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 `Csub`.

```
int bBegin = BLOCK_SIZE * bx;

int bStep = BLOCK_SIZE * wb;

int Csub = 0;
```

Matrix multiplication (11/16)

- The main loop starts by copying the working tiles of A and B to shared memory.

```
for(int a = aBegin, b = bBegin;  a <= aEnd; a += aStep, b += bS
    // 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();
```

Matrix multiplication (12/16)

- 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 computa
    // done before loading two new tiles of A dnd B in the
    __syncthreads();
}
```

Matrix multiplication (13/16)

- Once computed, the working tile of C is written to global memory.

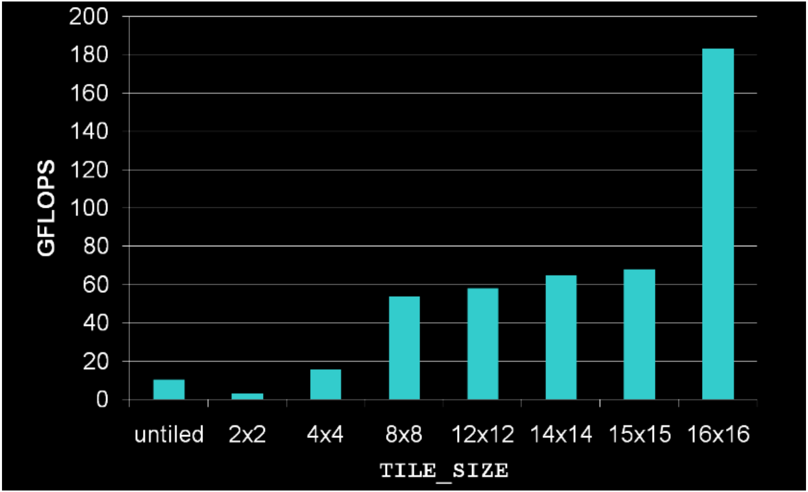
```
// Write the working tile of $C$ to global memory;
// each thread writes one element
int c = wb * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wb * ty + tx] = Csub;
```

Matrix multiplication (14/16)

- Each thread block should have many threads:
  - $\text{TILE\_WIDTH} = 16$ implies $16 \times 16 = 256$ 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 $256 \times (2 \times 16) \quad = \quad 8,192 \text{ fp ops}$:
  - 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.

Resource	Per GT200 SM	Full Occupancy on GT200
Registers	16384	<= 16384 / 768 threads = 21 per thread
`__shared__` Memory	16KB	<= 16KB / 8 blocks = 2KB per block

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

A simple copy kernel (1/2)

```
__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 (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.

```
__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

```
__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:

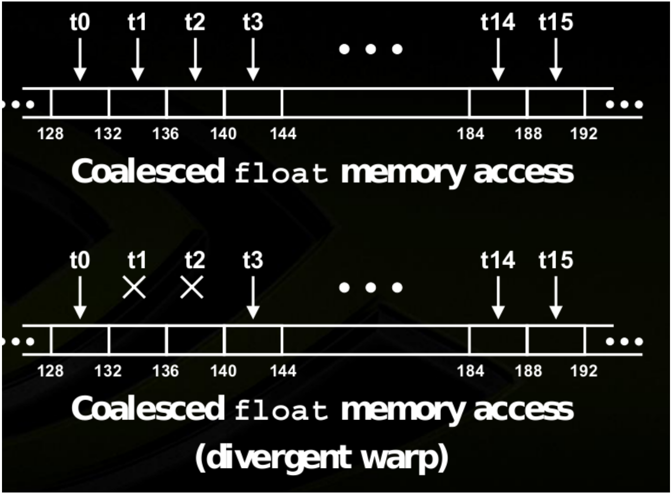
Routine	Bandwidth (GB/s)
copy	105.14
naive transpose	18.82

The minor differences in code between the copy and nave transpose kernels have a profound effect on performance.

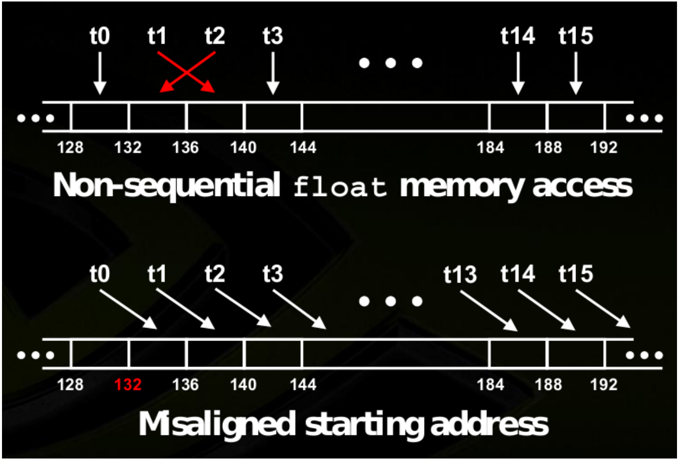
Coalesced Transpose (1/11)

- 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**)

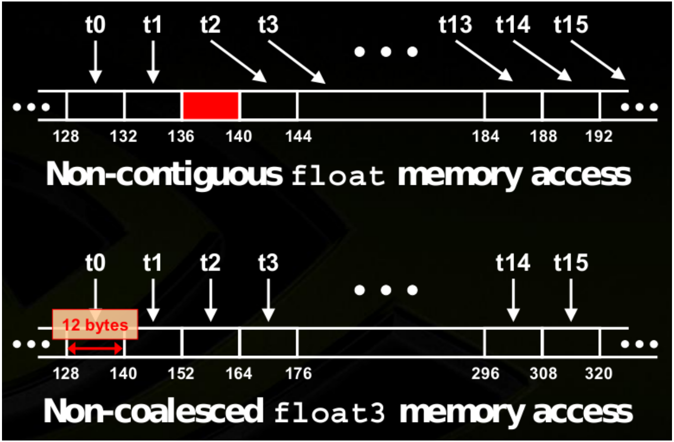
Coalesced Transpose (2/11)



Coalesced 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.

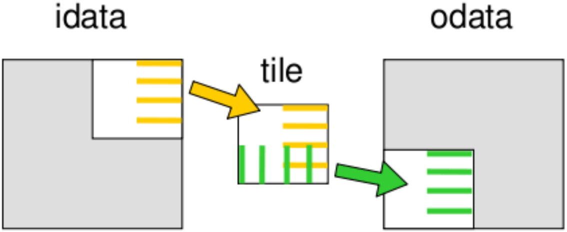
Coalesced Transpose (6/11)

- 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 `__synchthreads()` 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)

```
__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];
} }
```

Coalesced Transpose (8/11)



1. 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.
2. After a `__syncthreads()` call to ensure all writes to tile are completed,
3. 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)

Routine	Bandwidth (GB/s)
copy	105.14
shared memory copy	104.49
naive transpose	18.82

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)

```
  _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)

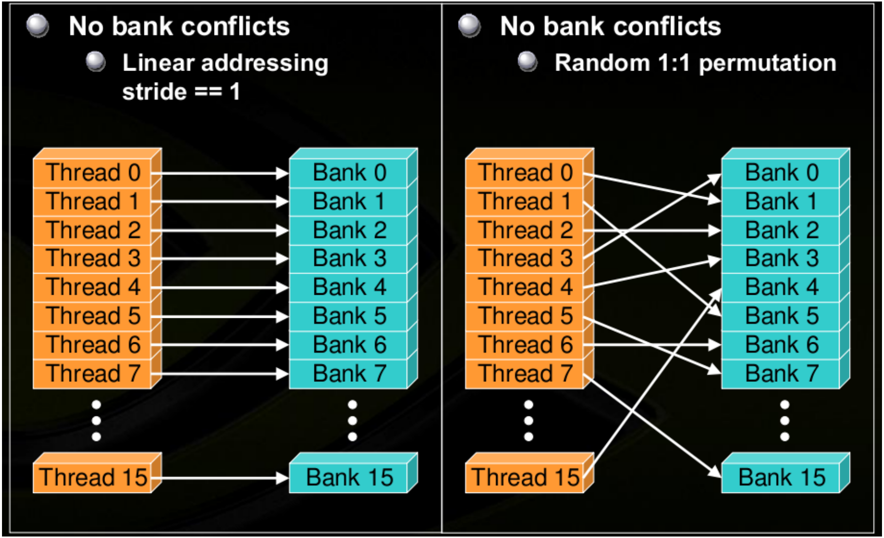
Routine	Bandwidth (GB/s)
copy	105.14
shared memory copy	104.49
naive transpose	18.82
coalesced transpose	51.42

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.

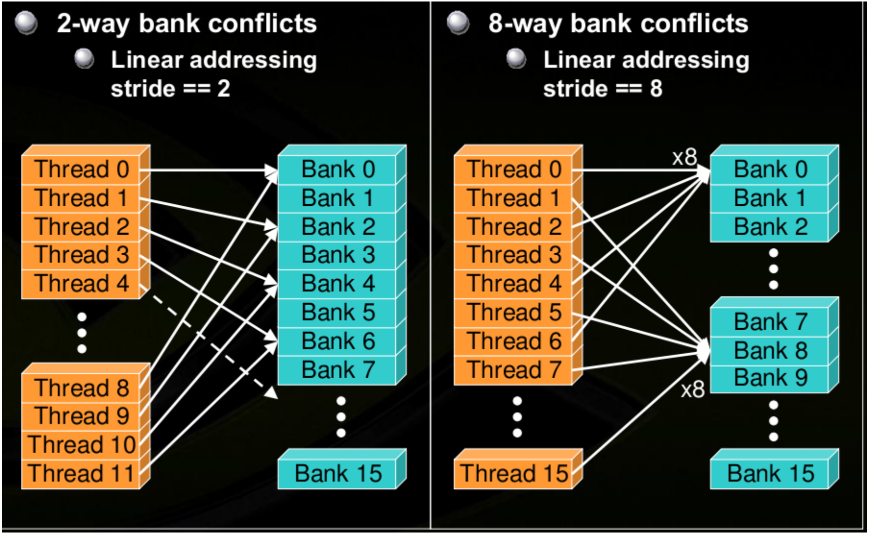
Shared memory bank conflicts (1/6)

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)



Shared memory bank conflicts (4/6)

1. The coalesced transpose uses a 32×32 shared memory array of floats.
2. For this sized array, all data in columns `k` and `k+16` are mapped to the same bank.
3. 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.
4. A simple way to avoid this conflict is to pad the shared memory array by one column:

```
__shared__ float tile[TILE_DIM][TILE_DIM+1];
```

Shared memory bank conflicts (5/6)

- 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.

Shared memory bank conflicts (6/6)

```
Device : Tesla M2050
Matrix size: 1024 1024, Block size: 32 8, Tile size: 32 32
                 Routine           Bandwidth (GB/s)
                    copy                105.14
      shared memory copy                104.49
         naive transpose                 18.82
     coalesced transpose                 51.42
 conflict-free transpose                 99.83
```

- While padding the shared memory array did eliminate shared memory bank conflicts, as was confirmed by checking the `warp_serialize` 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

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

Expose 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 $<->$ GPU Data Transfers

- CPU $<->$ GPU memory bandwidth much lower than GPU memory bandwidth
- Minimize CPU $<->$ 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

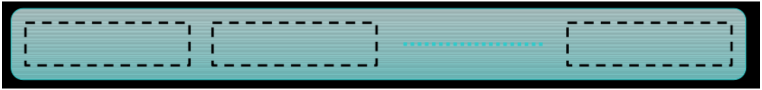
1. Partition data into subsets that fit into shared memory
2. Handle each data subset with one thread block
3. Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.
4. Perform the computation on the subset from shared memory.
5. Copy the result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

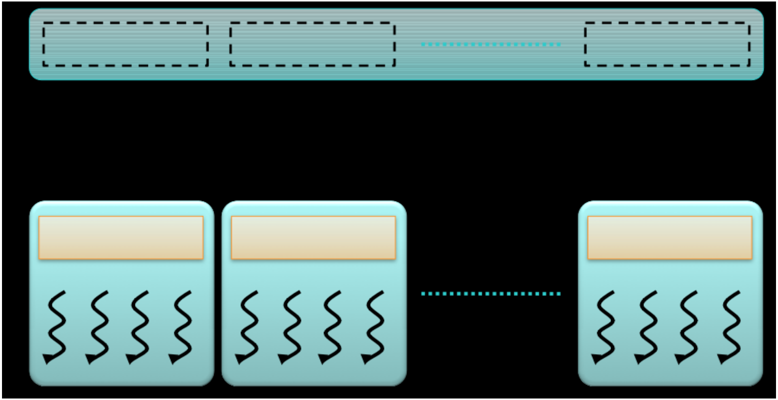
A Common Programming Strategy

Partition data into subsets that fit into shared memory



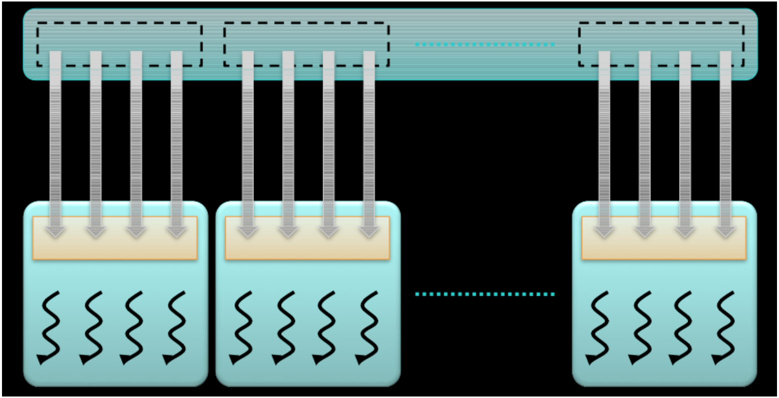
A Common Programming Strategy

Handle each data subset with one thread block



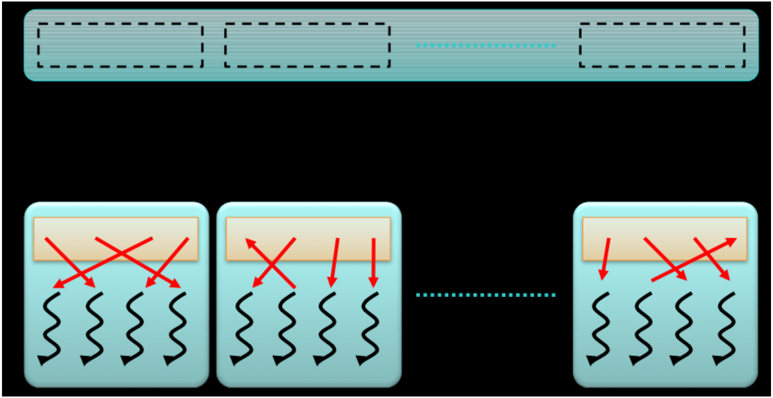
A Common Programming Strategy

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.