CS3211

AY23/24 Sem 1

github.com/SeekSaveServe

Lectures

Introduction

L0 and L1

Program Parallelization

Decomposition: Decompose a sequential algorithm into tasks (programmer)

- Granularity of tasks are important
- Tasks have dependencies (data or control) between each other which defines the execution order

Scheduling: Assign tasks to processes (programmer / compiler)

Mapping - Map processes to cores (OS)

Von Neumann Computation Model instruction and data are stored in memory, and processors computes.

Memory Wall disparity between memory speed and processor speed ($\leq$ 1 ns VS $\geq$ 100 ns)

Processing unit refers to a core that can execute a kernel thread

Interconnect busses betwen different components in the machine

Node Machine in a distributed system

Why Parallel

Primary Reasons

1 OVercome limits of serial computing
2 Solve larger problems
3 Save (wall-clock) time

Other Reasons

- Take advantage of non-local resources
- Cost/energy saving - use multiple cheaper computing resourcees
- Overcome memory constraints

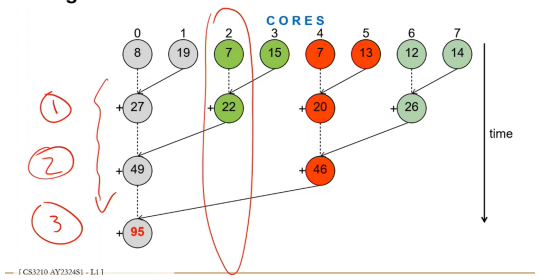
Computational Model Attributes

- **Operation mechanism** Primitive units of computation or basic actions of the computer on a specific Architecture
- **Data Mechanism** How we access and store data in address space
- **Control Mechanism** How primtive units of computation are scheduled
- **Communication Mechanism** Modes and patterns of exchanging information between parallel tasks (e.g message passing, shared memory)
- **Synchronization Mechanism** ensures to ensure needed information arrives at the right time

Dependencies and Coordination

- Dependencies among tasks impose constraints on scheduling
- Memory organizations: Shared-memory (threads), distributed-memory (processes)
- Coordination (synchronisation) imposes additional overheads

Two algorithms



- Core 0 is active throughout the execution
- Some cores are idle
- This is a lot better than having all cores idle while the master core is executing

Parallel Performance

- Execution time Vs Throughput
- Parallel execution time = computation time + parallelization overheads
- Overheads: Distribution of work(tasks) to porocesses, information exchange, synchronisation, idle time, etc

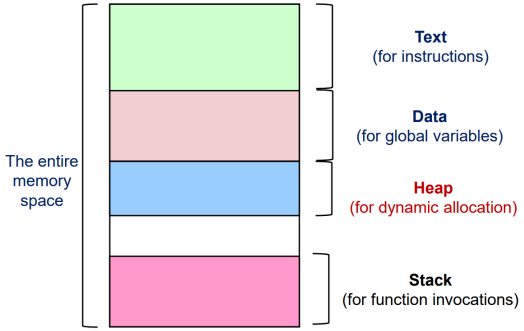
Background on Parallelism

L2: Processes and Threads

Process

- Identified by PID
- Program counter, global data (open files, network connections), stack or heap, current values of the registers (GPRs and Special)
- These information are abstracted in the PCB, and each proecss can be viewed as having exclusive access to tis address space
- Explicit communication is needed
- **Disadvantage**

1. High overhead of system calls
2. Potential re-allocation of data-structures
3. Communication goes through OS (system calls) and context switch is costly



Multi tasking

- Overhead: Context switching (PCB change) is needed and states of suspended process must be saved
- Time slicing: Pseudo-parallelism
- Child processes can use parent's data

Inter-process communication (IPC)

- Shared memory: need to protect access with locks
- Message passing: Blocking, unblocking, Synchronous, unsynchronous

Exceptions

- Executing a **machine level instruction** can cause exception
- For example: Overflow, Underflow, Division by Zero, Illegal memory address, Mis-aligned memory access

Interrupts

- External events can interrupt the execution of a program
- Usually hardware related: Timer, Mouse Movement, Keyboard Pressed etc

- **Synchronous**
  - Occur due to program execution
  - Have to execute an **exception handler**
- **Asynchronous**
  - Occur **independently** of program execution
  - Have to execute an **interrupt handler**

Threads

- A process may have multiple indepedent control flows called threads
- Each thread has its own stack and registers (PC, SP, registers), but share the same address space
- Shared memory model and Shared memory architecture
- Faster thread generation- no copy of address space
- Different process can be assigned to run on different cores of a multicore processor
- **User threads**
  - Managed by library
  - Context siwtch is fast, OS not involved
  - **Disadvantage**
    1. OS cannot map different threads of the same process to different resources $\Rightarrow$ No parallelism
    2. OS cannot switch to another thread if one thread blocks
- **Kernel threads**
- OS is aware of the threads and can manage accordingly
- Efficient in a multicore system
- Potential synchornisation issues

Many to one mapping

- All user-level threads mapped to one process.
- Efficiency depends on threading library

One to one mapping

- Each user-level thread is mapped to one kernel thread
- OS schedules

Many to many mapping

- Many user-level threads mapped to many kernel threads
- Library threads has overheads, and kernel threads has overheads
- At different points in time, different user threads are mapped to different kernel threads
- Number of threads must be suitable to the degree of parallism and the resources available

Locks

- Spinlock: busy wait
- Blocking: mutex
- Using more locks increasese the number of context switches
- DO NOT wait in the critical section

Semaphores

- Essentially shared global variables
- Can be potentially accessed anywhere in program
- No connection between semaphone and the data being protected

Barrier

- All threads must reach the barrier before any thread can proceed

Deadlock

- Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set
- **iff these conds are met**

1. Mutual exclusion-at least one resource is not shareable
2. Hold and wait - at least one process holding a resource and waiting for another
3. No preemption - crticial section cannot be aborted externally
4. Circular wait

- **Dealing with deadlock**
- Ignore it, prevent it, avoid it by controlling resource allocation, detection and recovery by breaking cycles

Starvation

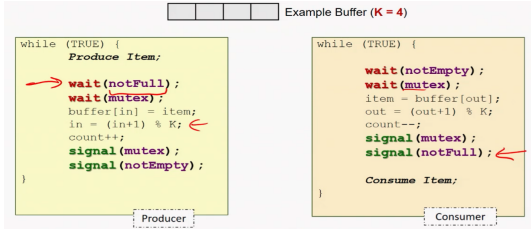
- Side effect of the scheduling algorithm. Lower priority processes might starve

Livelock

- Active acquire release but no useful work done

Producer-Consumer Problem

- Specifications:
  - Producers put in a shared bounded buffer if not full, consumers read from it if not empty
- Solution:



- Concurrent read, exclusive write. Categorical starvation of writer is possible

```
Writers                          Readers
roomEmpty.wait ()                mutex.wait ()
    #critical section for writers readers += 1
roomEmpty.signal ()              if readers == 1:
                                     roomEmpty.wait () # first in locks
                                 mutex.signal ()
                                 # critical section for readers

                                 readers -= 1
                                 if readers == 0:
                                     roomEmpty.signal () # last out unlocks
```

- Light switch: Abstracts out the shared lock for the reader

```
counter = 0
mutex = Semaphore (1)
lock (semaphore):
    mutex.wait ()
    counter += 1
    if counter == 1:
        semaphore.wait ()
    mutex.signal ()
unlock (semaphore):
    mutex.wait ()
    counter -= 1
    if counter == 0:
        semaphore.signal ()
    mutex.signal ()
```

- Starvation free solution (block out readers):

Writers	Readers
turnstile.wait ()	turnstile.wait ()
roomEmpty.wait ()	turnstile.signal ()
# critical section for writers	readSwitch.lock (roomEmpty)
turnstile.signal ()	# critical section for readers
roomEmpty.signal ()	readSwitch.unlock (roomEmpty)

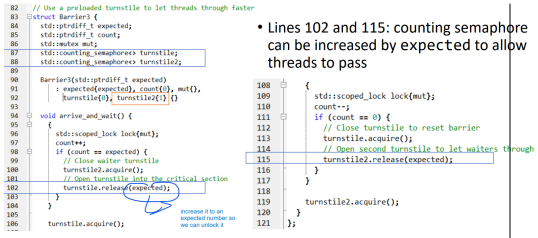
- Prioritise Writer:

```
readSwitch = lightswitch();
writeSwitch = lightswitch();
noReaders = semaphore(1);
noWriters = semaphore(1);

reader() {
/* Waiting for writers to be done */
wait(noReaders);
/* Writers cannot enter */
    readswitch.lock(noWriters);
signal(noReaders);
    # critical section
readSwitch.unlock(noWriters);
}

writer() {
/* Immediately acquires no readers so
    writers have priority */
writeSwitch.lock(noReaders);
    wait(noWriters);
    # critical section
    signal(noWriters);
writeSwitch.unlock(noReaders);
}
```

- This is implemented in C++ as a **shared_lock** and **unique_lock**
- GO has something similar: readLock and writeLock

Barrier

- All threads must stop at a common point before proceeding, can be reusable (barrier) or single use (latch)
- std::barrier, std::latch in C++
- E.g. std::barrier arrivalPoint(size) ... arrivalPoint.arrive_and_wait()
- sync.WaitGroup in GO is a latch, we can use 2 of them to make a barrier
- C++ implementation
  - The naive version fails because context switch can happen right before counter == N, which causes multiple threads (that were context switched out after counter ++) to signal the switch (another way to fail is to have 1 thread lap everyone else between the first barrier unlocks and second barrier unlocks)
  - The solution is to add a second turnstile (initialised as 1) to guard the turnstile1.signal, such that only one thread can signal it
  - But using mutex to increment turnstile1 one-by-one is slow
  - So we use a counting semaphore instead so we can raise the barrier by 1 thread!



- Lines 102 and 115: counting semaphore can be increased by expected to allow threads to pass

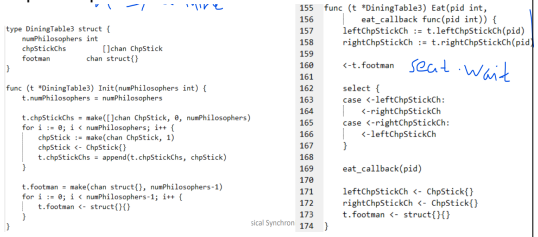
Dining Philosophers

- Specifications: N philosophers, N chopsticks
- Deadlock: All pick up left simultaneously
- Livelock: Put down left if right cannot be acquired
- Slap a mutex: Becomes sequential
- Scoped Lock(left, right): Acquire multiple mutexes in a deadlock free manner (deadlock avoidance), but as we have seen in CS3223, deadlock avoidance can lead to livelock
- GO's Mutex Free Solution: Use odd-even ring communication, odd numbered philosophers pick up left first, even numbered philosophers pick up right first
- This is the same as the right hander argument
- Tanenbaum's solution:

```
#define N 5
#define LEFT ((i+N-1)% N)
#define RIGHT ((i+1) % N)
#define THINKING 0
#define HUNGRY 1
#define EATING 2
int state[N];
Semaphore mutex = 1;
Semaphore s[N];

void philosopher( int i ){
    while (TRUE){
        Think( );
        takeChpStcks( i );
        Eat( );
        putChpStcks( i );
    }
}
void safeToEat(i)
{
    if( (state[i] == HUNGRY) &&
    (state[LEFT] != EATING) &&
    (state[RIGHT] != EATING) ) {
    state[i] = EATING;
    signal(s[i]);
    }
}

void takeChpSticks( i )
{
    wait(mutex);
    state[i] = HUNGRY;
    safeToEat(i);
    signal(mutex);
    wait(s[i]);
}
```

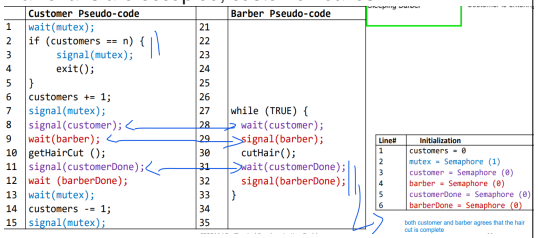
```
void putChpSticks(i)
{
    wait(mutex);
    state[i] = THINKING;
    safeToEat(LEFT);
    safeToEat(RIGHT);
    signal(mutex);
}
```

- Limited seats: Use a semaphore(N-1) to limit the number of philosophers that can eat at the same time

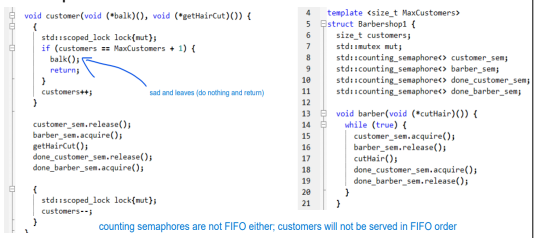


Barber Shop

- Barbershop consists of a waiting room with n chairs and the barber chair
- If there are no customers to be served, the barber goes to sleep
- If the barber is busy, but waiting room is available, customer seats on one of the chairs
- If barber is sleeping, customer wakes him up
- If all chairs are occupied, customer leaves



- line 31-32 are important since we need customer and barber need to agree that the haircut is done
- GO implementation:



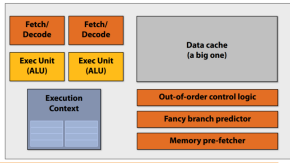
Architecture

L3: Processor and memory organization

Single Processor Parallelism

- Bit level - we work with word (multiple bits), data parallelism
- Instruction level (from same thread)

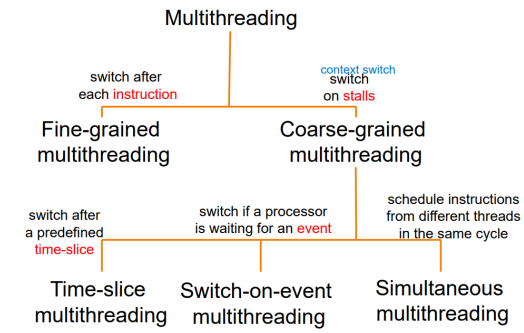
1. Pipelining - parallelism across time
   - Multiple instructions to occupy different stages in the same clock cycle - assuming no control or data dependencis
   - **Disadvanatges**
     1.1. Independence
     1.2. Bubbles - idle stages
     1.3. Data and control flow hazard
   - Wrong speculation of if-else branches can lead to wasted cycles
   - Synchronisation - need to preserve read-after-write
   - no more benefit to improving ILP now
2. Superscalar - parallelism across space



   - Duplicate pipelines and allow multiple instructions to pass through the same stage
   - Scheduling tough - which ones to execute together?
   - E.g Multiple ALUs
     2.1. Static - compiler decides
     2.2. Dynamic - hardware decides
- Thread level
  - Motivated by the limitation of ILP
  - SMT: Duplicate hardware context (PC, registers etc)
  - By convention, SMT is limited to 2 threads to reduce overhead and memory contention
  - Logical cores: hyperthreads

Processor level parallism

- Add more cores to processors to enable **multiple execution flows**
- Each core can be hyperthreaded
- Shared Memory
- Distributed Memory



Flynn's Taxonomy

- Describes parallel architecture based on instruction stream (execution flow - PC) and data stream

Single Instruction Single Data

- Single stream of instructions with each working on a single data
- Not to be confused with SIMD from parallel patterns

Single Instruction Multiple Data

- Single stream of instructions with each working on multiple data

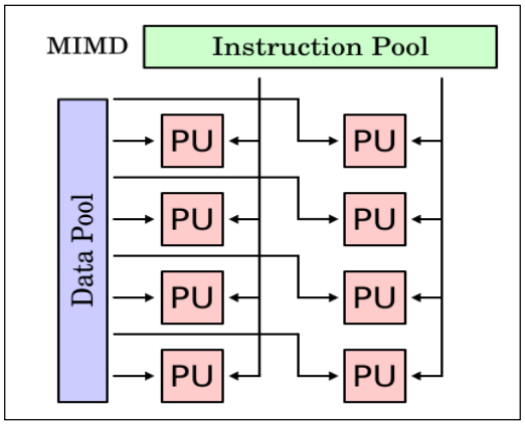
- Exploit data parallelism (vector processor)
- Same instruction broadcasted to all ALUs
- AVX: intrinsic functions operatie on vectors of 4 64 bit values

Multiple Instruction Single Data

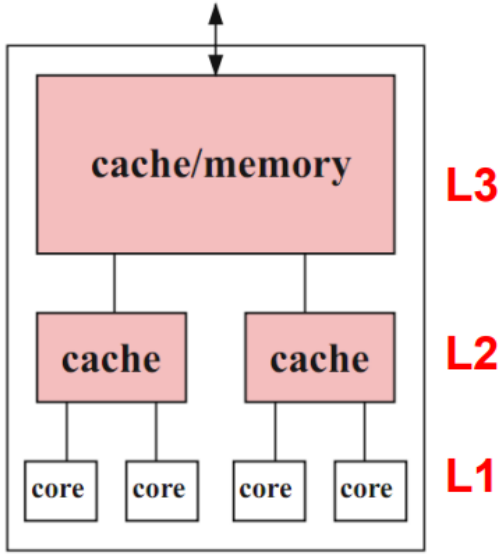
- Multiple instructions operating with a single data

Multiple Instructions Multiple Data

- Each PU fetches its own instructions
- Each PU operates its own data
-

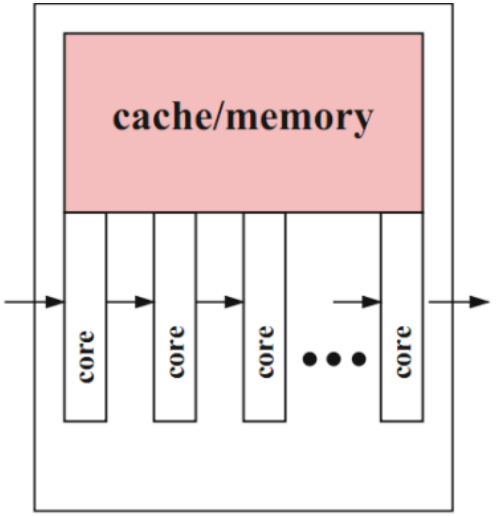


Hierarchical designs



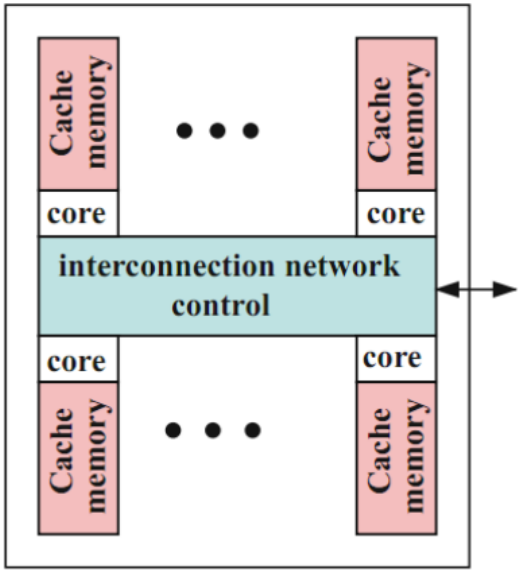
- Each core can have a separate L1 cache and shares the L2 cache
- All cores share common external memory

Pipelined design



- Multiple packets being processed in a pipelined fashion
- Cores connected linearly, shares the same cache, memory
- Useful if the same computation has to be applied to a long sequence of data elements

Network-based design



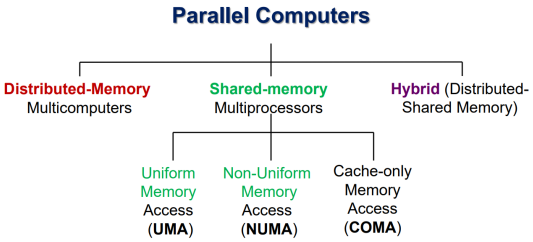
- Cores and their local memory and memories are connected via an interconnection network

Why cache

- cache provides high bandwidth data transfer to CPU and reduce latency in data access
- Memory latency: Amount of time for a memory request from a procesor to be serviced
- Bandwitdth: Rate at which the memory system can provide data to a processor
- A stall happens when the next instruction depends on previous instructions
- Bandwidth and latency affects stalls, since instructions (sw, lw) needs to wait for the memory system to become available

Performant parallel programs

- Try not to overload the memory system with too many requests
- Share data across threads (inter-thread cooperation)
- Reuse data fetched previously (temporal locality)
- **Favor additional arithmatic over load / store**



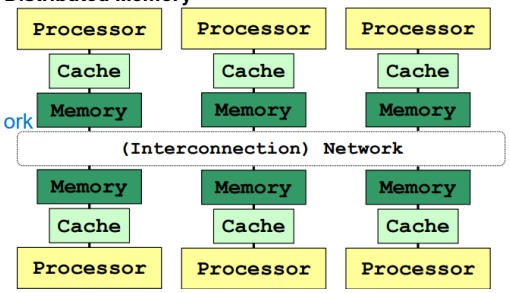
Cache coherence

- Multiple copies of data exist on different caches
- Local updates should not be seen by other processes
- Maintained by additional insructions
- Instructions that mess up cache coherence hence presents severe overheads

Memory consistency

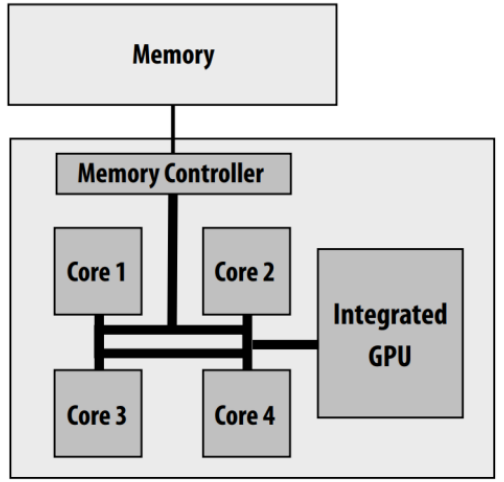
- Memory consistency depends on the PL and architecture
- A seq consistent architecture makes a PL with seq const memory model run faster since fewer instructions are needed to ensure memory consistency

Distributed Memory



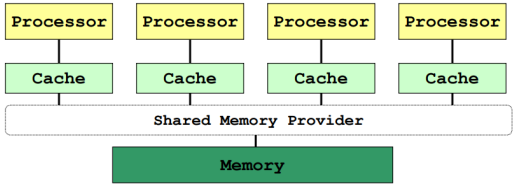
- Each node is an independent unit with processor and memory
- Memory in each node is private
- Nodes communicate through a network

Shared memory



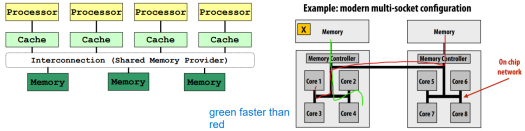
Intel Core i7 (quad core) (interconnect is a ring)

- Parallel programmes share memory through controller / provider
- Cache coherence and memory consistency is ensured



Uniform Memory Access

- Latency of accessing main memory is the same for processors
- Suitable for samll number of processors. Contention over memory can be high for large number of processes

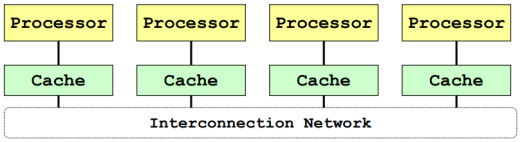


Non-uniform Memroy Access

- Physically distributed memory of all processing elements are combined to form a global shared memory
- Local memory access has lower latency
- Reduce contention since each processor tend to access local memory
- Adding more processes does not increase contention as much as UMA
- Data consistency is easier too

Cache Coherent NUMA (CCNUMA)

- EAch node has cache to reduce contention



Cache only Memory Architecture (COMA)
- Each memory blocks works as cache memory. This means that no fixed space stores data permanently and cache block with data can be moved around dynamically.
- Data migrates dynamically to keep data as close as possible to the processors
- Cache coherence is harder since data may not just be copied, they can also be shifted around.

L7: Cache coherence and memory consistency

Cache properties
- Larger cache reduces cache miss but increases access time
- Block size (cache line): data is transferred between main memory and cache in blocks of fixed size
- Larger block size – greater spatial locality
- Smaller block size – shorter replacement

Case Study: Matrix Multiplication
- Size of matrix: A 256KB cache can only store a matrix of floats of size (178*178) * 8 Bytes (float size)

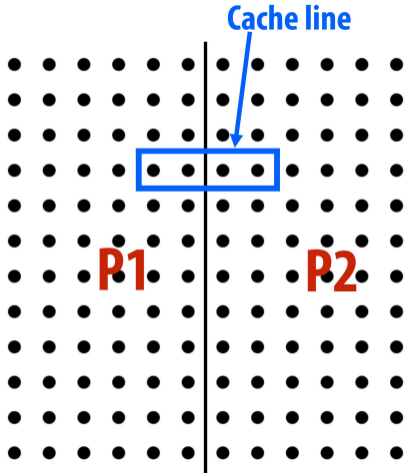
Write Policy
- **Write through**
  - Write access is immediately transferred to memory
  - Advantage: always get the newest value of a bloc
  - Disadvanatge: slow down due to many memory access (use a buffer!)
- **Write-back**

Line state	Tag	Data (64 bytes)

  - Write is only performed in the cache, write to main memory is only performed when the cache is replaced (dirty bit)
  - Advantage: fewer write operations
  - Disadvantage: memory may contain invalid entries

Cache coherence
- Problem: Multiple copies of the data exists on different cache lines, stale data may exist
- **Coherence**
  - Each processing unit should have a consistent view of the memory through its local cache
  - All processing units should agree on the order of read writes to the same memory space
  - Property 1: Program Order Property
    - Programme should observe the effects of writes in the order of the programme
  - Property 2: Write propagation
    - Writes become visible to other processing units eventually
  - Property 3: Write serialization
    - Given:
      1. write v_1 to x
      2. write v_2 to x
    - programme should never read x as v_2 and then as v_1
    - All writes to a location are seen in the same order by all execution units, eventually

Tracking cache line sharing status
- **Snopping based**
  - No centralised directory
  - Each cache keeps track of the sharing status
  - Cache monitors and snoop on the bus to keep the cache line updated
  - Used in architectures with a bus
  - Write Propagation: All the processing units on the bus can observe changes made by every other bus
  - Write serialization: Bus transactions are visible to the processing units in the same order
  - Granularity: cache block
- **Directory based**
  - Sharing status is kept in a central directory
  - Commonly used in a NUMA architecture
- **Implications**
  - Increased in overhead: increased memory latency, reduced cache hit rate
  - Cache ping-pong: the effect where a cache line is transferred between multiple cores as a result of true / false sharing
  - False sharing: different threads have data that is not shared in the program, but this data gets mapped to the same cache line
  - False sharing makes cache ping pong difficult to detect, since the code ensures that memory are not shared but they happened to be mapped to the same cache line



Memory Consistency Models
- Coherence ensures that processing units agree on the order of writes on the SAME memory location, and that all writes to shared memory will eventually propagate
- Consistency ensures that processing units agree on the order of writes on DIFFERENT memory locations
- Under the consistency rules, the instructions can be reordered to hide latencies
- **4 types of memory operations orderings**
  - must commit – the results are visible
  - W $\rightarrow$ R: write to X must commit before the subeqeuqnt read of Y
  - R $\rightarrow$ W: read of X must commit before the subsequent write of Y
  - R $\rightarrow$ R: read of X must commit before the subsequent read of Y
  - W $\rightarrow$ W: write to X must commit before the subsequent write of Y

Sequential Consistency
- Every processing unit isues their memory operations in programme order
- Global results of all memory operation on every memory address appear in the same sequential order to evry processing unit
- All 4 memory operation orderings are observed
- Poor performance
- Examples:

processor	P_1	P_2	P_3
program	(1) $x_1 = 1$;	(3) $x_2 = 1$;	(5) $x_3 = 1$;
	(2) print x_2, x_3;	(4) print x_1, x_3;	(6) print x_1, x_2;

  - **Once 1 core sees an interleaving, the same interleaving will be observed by other cores**
  - Possible interleavings
  1. (1)-(3)-(5)-(2)-(4)-(6)
  2. (1)-(2)-(3)-(4)-(5)-(6)
  - Impossible output: 011001
  - To produce 0110, we need something like (1)-(3)-(2)-(4). But after this it is not possible to produce another 0 since the last read statements happens after all the write statements

Relaxed memory consistency
- Relax if data dependencies allow
- **Data dependency: if two ops access the SAME memory location**
  - R $\rightarrow$ W
  - W $\rightarrow$ W
  - W $\rightarrow$ R

Relaxed Consistency: Write-to-read (WR)
- Allows a read on processing unit P to be reorderd wrt the previous write operations on different memory locations
- Data dependencies must be observed, but it is only wrt the same memory location
- Data dependencies cannot be chained
- Different models depends on the timing of return
- **Total Store Ordering**
  - Processing units can **move its own reads** infront of its own writes
  - **Write Atomicity is observed**: Reads by other processing units cannot return new values of address A untill the write to A is observed by all PUs
- **Processor Consistency**
  - **Write atomicity is not observed**: write can be read by some processing units before they are read by other processing units
  - **Write serialization is observed**: writes to the same memory location are seen in the same order by all processing units

SC:
W -> R
R -> R
R -> W
W -> W

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

PSO:
~~W -> R~~
R -> R
R -> W
~~W -> W~~

Relaxed Consistency: Write-to-Write (WW)
- Writes can bypass earlier writes (to different locations) in write buffer
- Allows write misses to overwrite to hide latency
- Can only reorder within the same processing unit
- **Partial Store Order**
  - Relax W $\rightarrow$ R similar to TSO
  - Relax W $\rightarrow$ W
- **Example 1**

P1
```
A = 1
B = 1
```
P2
```
print B
print A
```

  - Only PSO can observe A=0, B=1 since only it reorders WW
- **Example 2**

P1
```
A = 1
```
P2
```
while ( A == 0 );
B = 1
```
P3
```
while ( B == 0 );
print A
```

  - TSO and SC cannot observe A=0 due to write atomicity
  - PC can observe A=0, since observing B=1 does not mena that it has observed A=1 (not write wtomicity)
  - PSO cannot observe (0,0) since it still observes write atomicity

Property	Sequential Consistency (SC)	Relaxed Consistency: Total Store Ordering (TSO)	Relaxed Consistency: Processor Consistency (PC)	Relaxed Consistency: Partial Store Ordering (PSO)
Respects data dependencies within the same core (e.g., don't touch: x = 5, read x)	Yes			
Preserves R → R and R → W order	Yes			
Preserves W → R	Yes	No	No	No
Preserves W → W	Yes	Yes	Yes	No
All processors must be able to see same value before a read completes? (Write Atomicity)	Yes	Yes	No	Yes

L11: Interconnection networks

Parallel Computation Models

L4: Shared-memory programming models

Parallelism
- Average number of units of work that can be performed in parallel per unit time.

- E.g. MIPS, MFLOPS
- Limitation: Program dependencies - data, control
- Runtime delays - memory contention, communication overheads, thread overhead, synchronisation
- We cannot reorder them however we like
- Work = Task + dependencies (limitations)

Data parallelism

- If iterations are **independent**, they can be executed in arbitrary order on multiple cores
- Patition data among processing units, each doing similar work
- Commonly expressed as a loop, if the iterations are independent and can be executed in arbitrary order
- E.g. SIMD computers
- **OpenMP - matrix multiplication**

```
// parallelize result = a x b
// each thread works on one itreration of
     the outer-most loop
// vars (a, b ,result) are shared
#pragma omp parallel for num_thread(8)
    shared(a, b, result) private(i, j ,k)
    ...
```

- **Same as**

```
for (i=0; i < size; i ++)
    for (j=0; j < size; j ++)
        for (k=0; k < size, k ++)
            result[element][i][j] +=
                a.element[i][k] *
                b.element[k][j]
```

- **Single Program Multiple Data (SPMD)**
  - Same programme may behave differently based on the data
  - Good if
  - E.g. Scalar product of $x.y$ on p processing units

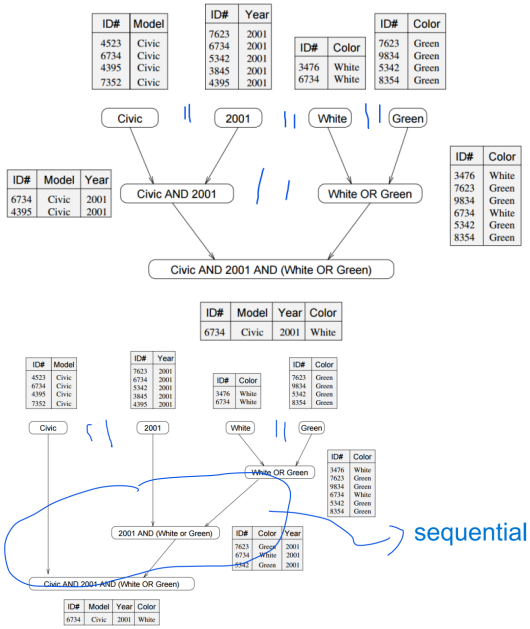
```
local_size = size/p;
local_lower = me * local_size;
local_upper = (me+1) * local_size - 1;
local_sum = 0.0;

for (i=local_lower; i<=local_upper; i++)
  local_sum += x[i] * y[i];

Reduce(&local_sum, &global_sum, 0, SUM);
```

Same program executed by **p** processing units.

"me" is the processing units index (0 to p-1)

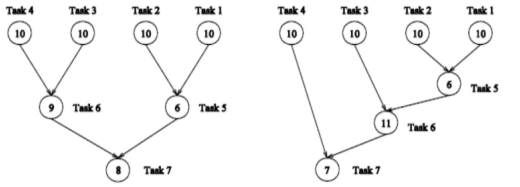
Task parallelism

- Partition the tasks among the processing units
- indepedent program tasks/ parts can be executed in parallel
- Granularity: statement, loop, function
- More complexed than data parallelism → needs to schedule, map, take care of dependencies ...
- **Decomposition**
  - The room for parallelism in a task depends on how the task is decomposed



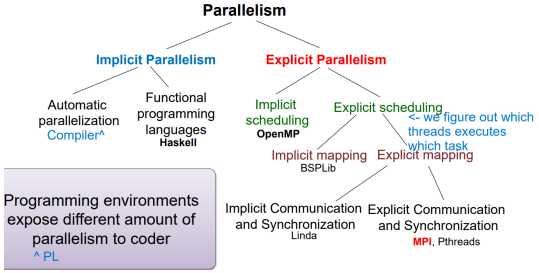
Task dependence graph

- DAG: node=tasks, value=expected execution time, edge=condtrol dependency
- Bad for one process to take disproportionately more data → idle time
- Critical path length: maximum slowest completion time
- Degree of concurrency=total work/critical path length



Critical Path = (Task 4 → 6 → 7)
Critical Path Length = **27**
Degree of concurrency = 63 / 27 = **2.33**

Critical Path = (Task 1 → 5 → 6 → 7)
Critical Path Length = **34**
Degree of concurrency = 64 / 34 = **1.88**



Coordination: Shared memory

- Protect access to shared address space, mutex.
- Needs hardware support to implement efficiency. NUMA makes it easier but it is still costly to scale due to contention (any processor can load/ store to any address)
- Can be done without a shared memory system (NUMA, UMA)
- Any type of coordination can be used in any hardware via software

Coordination: Data-parallel

- SIMD, vector processors
- Traditional: Map a function onto a large collection of data
- Side effect free execution
- Modern: Data-parallel languages do not enforce this structure
- SPMD model used in CUDA, OpenCL, ISPC instead

Coordination: Message passing

- Tasks operate within their own private address space and communicate by explicitly sending / receiving messages
- E.g. MPI, GO
- Hardware does not implement system wide loads and stores, can connect commodity systems toegther to form large parallel machines
- Many many computers, not a very big one
- Compatible with distributed memory systems

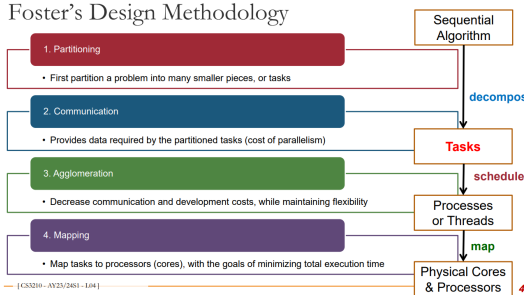
Coordination and hardware

- Shared memory: UMA, NUMA. Copies of messages and sent / received from library buffers
- Message passing: distributed systems, clusters, supercomputers
- Any abstraction can be implemented with any hardware but it will be more costly
- Shared address space on incompatible hardware
  - Write: Send message to all cores to invalidate value
  - Read: page fault handler issues appropriate network requests

Summary of Coordination Models

- Shared address space: very little structure
  - All threads can read and write to all shared variables
  - Drawback: not all reads and writes have the same cost (and that cost is not apparent in program text)
- Data-parallel: very rigid computation structure
  - Programs perform the same function on different data elements in a collection
- Message passing: highly structured communication
  - All communication occurs in the form of messages

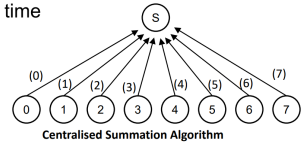
Foster's Design Methodology



Foster's Design methodology

1. Partitioning
   - Divide computation and data into independent pieces to discover maximum parallelism
   - Two approaches:
     1.1. Domain decomposition: divide data into smaller, equal pieces. Associate computation with data.
     1.2. E.g. 24 tasks with 3 grids each → 6 tasks with 12 grids each
     1.3. Functional decomposition: Divide computation into piece. Associate data with computation.
     1.4. E.g. Climate model → Atmospheric model, hydrology model ...
   - Rule of thumb:
     - 10x more primitive tasks than cores in target computer
     - Minimize redundant computations and redundant data storage
     - Primitive data should be of roughly the same size
     - Number of tasks an increasing function of problem size
2. Communication (coordination)
   - Dependencies between tasks necessitates communication
   - Overlap computation and communication such that when some tasks are communicating, others are computing (improve utilisation)
   - **Local Communication**
     - Tasks needs data from a small number of other tasks (neighbors)
     - Use channel
     - E.g. 2-D finite state computation (requires 5 points to compute next state)
   - **Global Communication**
     - Significant number of tasks contribute to perform a computation
     - Do not create channels early on in the execution
     - E.g. Unoptimised sum N numbers
       - Does not distribute computation and communication - **centralised**
       - Does not allow overlap of computation and communication - **Sequential**

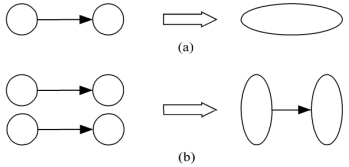
Unoptimized sum N numbers distributed among N (= 8) tasks need O(N) time



   - Rule of thumb:
     - Communication operation balanced among tasks
     - Each task communicates with only a small group of neighbors
     - Tasks can communicate in parallel
     - Overlap computation with communication
3. Agglomeration
   - Combine tasks into larger tasks s.t **tasks** $\geq$ **cores**
   - Goals:
     - Improve performance by reducing cost of task creation and communication
     - Maintain scalability of program
     - Simplify programming
   - E.g. Granular: One task per grid
     - 8*8=64 tasks
     - 64 * 4 (neighbors) * 2(send/ receive)=512 data transfers
   - Coarse: 16 grid per task
     - 2*2=4 tasks

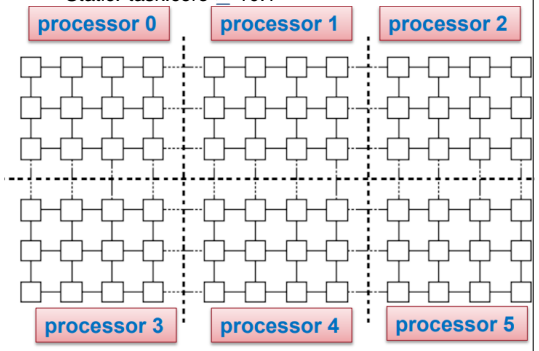
- 4*4*2=32 data transfers
- larger messages

- Rule of thumb:
  - Increases locality of parallel programmes (more neighbors read)
  - Number of tasks increases with problem size
  - Number of tasks suitable for likely target increases (10xnumCores)
  - Trade-off between agglomeration and code modification should be resonable (man hour)



Reduce number of sends and receives

4. Mapping
   - Assignment of tasks to execution units
   - Goals:
     - Maximise processor utilisation: place tasks of different cores
     - Minimise inter-process communication:Place tasks that communicate often on the same core to increase locality
   - Can be performed by user (distributed memory systems) or OS (centralised multiprocessor)
   - Rule of thumb:
     - Finding optimal mapping is NP hard in general (set cover)
     - Consider designs based on one task per core and multiple tasks per core
     - Evaluate static and dynamic task allocation
       - Dynamic: allocator should not be performance bottleneck
       - Static: task:core $\geq$ 10:1



(same amount of work on each processing unit and high locality)

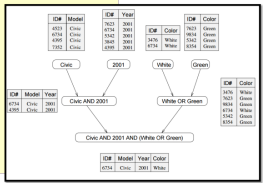
Automatic Parallelization
- Compilers perform decomposition and scheduling
- Drawbacks:
  - Dependence analysis is difficult for pointer-based computation or indirect addressing
  - Execution time of function calls or loops with unknown bounds is difficult to predict at compile time

Functional programming languages
- Describe the computations of a program as the evaluation of mathematical functions without side effects
- Advantage: New language constructs are not necessary to handle a parallel execution
- Challenge: Extract the parallelism at the right level of recursion

Parallel Programming Patterns
- Pattens are not mutually exclusive, use the best match

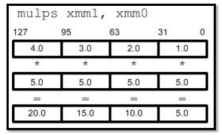
Fork Join
- Children run in parallel but are independent
- Children execute the same or different program
- Children join the parent at different points
- Good for loop parallelism (independent for loops)
- **Implementation:** Processes, threads etc

```
P1 = Fork {
    P3 = Fork { return Model = "civic" }
    P4 = Fork { return Year = "2001" }
    Join P3, P4
    Return P3 AND P4
}
P2 = Fork {
    P5 = Fork { return Color = "green" }
    P6 = Fork { return Color = "white" }
    Join P5, P6
    Return P5 OR P6
}
Join P1, P2
Return P1 AND P2
```



Parbegin - Parend
- most relaxed, code is structured into sequential segments and parallel segments
- Programmer specifies a sequence of statements to be executed in parallel
- A set of threads is created and the statement of the construct are assigned to these threads
- All the forks are done at the same time and all the joins are done at the same time
- Statements after parbegin and parend are only executed after all threads joins (barrier)
- **Implementation:** OpenMP or compiler directives
- E.g Matrix multiplication using openMD

SIMD (not the Architecture)
- Single instructions are executed synchronously by diferent threads on different data
- Similar to parbegin-parend but al threads execute the same instruction at the same time (synchronous)
- Parallel but synchronous
- **Implementation:** AVX / SSE instruction on intel processor



xmm registers are 128 bits long

SSE instruction treats the xmm registers as 4 individual 32-bit floating point value

SPMD
- Same program executed on different cores but operate o different data
- Different threads might execute on different instructions of the same program due to control flow (ifs) and speed of cores
- Similar to parbegin-parend but there is no implicit synchronization (lack of barrier)
- E.g. programs on GPGPU

Master-Worker
- Single program controls the execution of the program
- Master executes main function, assigns work to worker threads
- Initialisation, output and Coordination is done by master
- Worker waits for instruction
- **Benefit:** Good for simple and homogeneous worker threads and a master thread to organize them

```
int main(int argc, char ** argv)
{
    int nprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    size = 2048;
    // One master (rank = 0) and nprocs-1 workers
    if (myid == 0)  {
        master();
    } else {
        worker();
    }
    MPI_Finalize();
    return 0;
}
```

```
void master()
{
    matrix a, b, result;

    // Allocate memory for matrices
    allocate_matrix(&a);
    allocate_matrix(&b);
    allocate_matrix(&result);

    // Initialize matrix elements
    init_matrix(a);
    init_matrix(b);

    // Distribute data to workers
    master_distribute(a, b);

    // Gather results from workers
    master_receive_result(result);

    // Print the result matrix
    print_matrix(result);
}
```

```
void worker()
{
    int rows_per_worker = size / workers ;
    float row_a_buffer[rows_per_worker][size];
    matrix b;
    float result[rows_per_worker][size];

    // Receives data
    worker_receive_data(&b, row_a_buffer);

    // Performs computations
    worker_compute(b, row_a_buffer, result);

    // Sends the results to master
    worker_send_result(result);
}
```

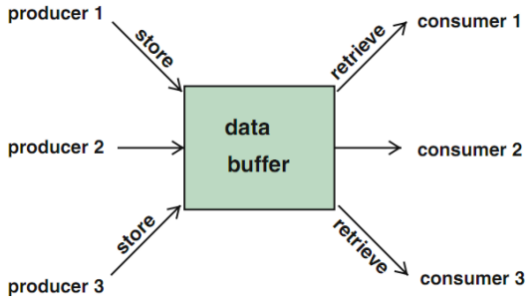
Task Pool
- Common data structure for threads to retrieve tasks
- Number of threads is fixed
- Threads are statically created by main
- Work is not pre-allocated. Instead worker retrieves new tasks from pool
- Thread can generate new tasks to put in pool and coordination is not done by master (difference from master-worker)
- May run into producer consumer issues when accessing the pool
- Execution is completed when the pool is empty AND each thread has terminated the processing of its last task
- **Benefits:**
  1. Adaptive can generate tasks dynamically, good for irregular applications
  2. Overhead for thread creation is independent from execution
- **Disadvanatges**
  1. For fine grained tasks, the overhead of retrieving and insertion becomes significant

```
class ThreadPoolExample {

    public static void main(String[] args) {
        ExecutorService executor =
                    Executors.newFixedThreadPool(5);

        for (int i = 0; i < 10; i++) {
            Runnable Task = new Task( ..... );
            executor.execute( Task );
        }
        ......
    }
}
```

5 threads

10 tasks added to the pool.

Producer Consumer

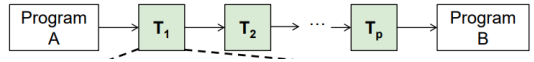


- Producer produces data which are used as input by consumer threads
- Synchronisation is needed to ensure correct coodination between producer and consumer threads

```
void produce() {
    synchronized (buffer) {
        while (buffer is full)
            buffer.wait();
        Store an item to buffer;
        if (buffer was empty)
            buffer.notify();
    }
}
```

```
void consume() {
    synchronized (buffer) {
        while (buffer is empty)
            buffer.wait();
        Retrieve an item from buffer;
        if (buffer was full)
            buffer.notify();
    }
}
```

Pipelining
- Data is aprtitioned into a stream that flows throuh pipeline stages synchronuously
- Each stage (threads) can be processed in parallel (functional parallel stream)



L6: Data parallel models (GPGPU)

Shader GPU
- Hard to transfer data between GPU and CPU
- No scatter: threads cannot write to arbitrary or multiple mem locations
- No communication between fragments
- Coarse thread synchronisation
- Example of data parallelism: fast processors for performing the same computation on large collection of data

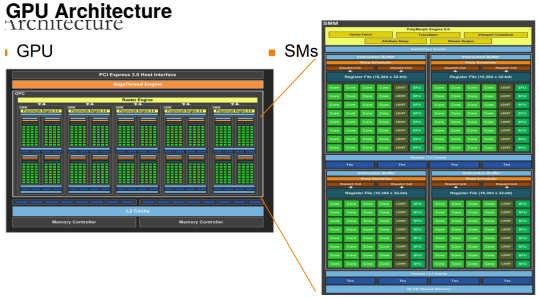
FLOPs performance on GPGPU

- Best performance with single precision FLOPs
- 2 processors need to work to perform double precision

GPU Architecture

Architecture

GPU SMs

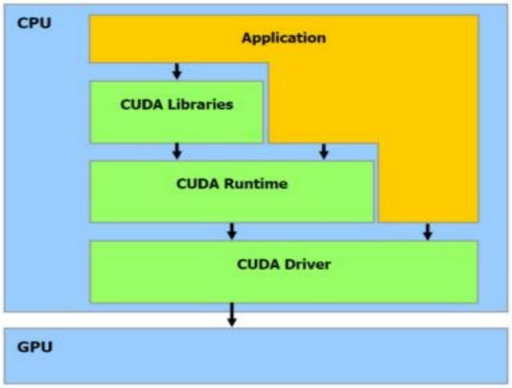


- Multiple Streaming Multiprocessors (SMs) - Memory, cache, connecting interface (PCI)
- SM consists of multiple compute cores
  - Memories(register, L1 cache, shared memory)
  - Logic for thread and instruction management

CUDA programming model

- Compute Unifified Device Architecture
- Simple extension to standard C
- Mature software stack (high-level to low level)
- User launches batches fo threads on the GPUFully general load / store memory model (CRCW)
- Scales with non-NVIDIA GPUs too
- Transparently scales to hundreds of cores and thousands of parallel threads
- Programmer focus on parallel algorithms
- Enable heterogeneous systems (CPU + GPU)

CUDA layers



CUDA kernels and threads

- Device=GPU
- Host=CPU
- Kernel=function that runs on the device
- Parallel portions execute on decice as kernels, and multiple are allowed in CUDA hardware
- CUDA threads are extermely light wait with minimal creation overhead and instant context switches
- The key is to divide work to thousands of threads

Arrays of parallel threads

- A CUDA kernel is executed by an array of threads
- All threads run the same code (SPMD)
- Each thread has an ID that is used to compute memory addresses and make control decisions

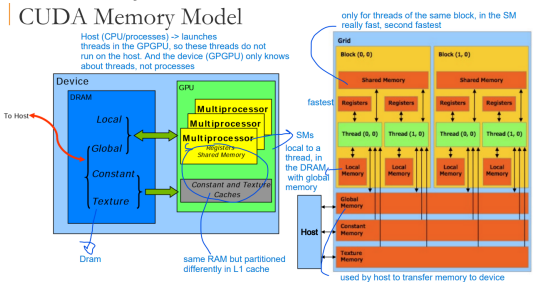
Thread cooperation

- Threads in the array need not be completely independent
- Shares results to save computation
- Share memory accesses which reduces bandwidth
- **Scalable Cooperation**
  - Divide monolithic thread awray into multiple blocks
  - In a block: shared memory, atomic operations and barrier synchronisation
  - Threads in different blocks cannot cooperate
- Enables programs to transparently scale to any number of proceses

Thread Execution Mapping to Architecture

- SIMT execution model
- Multiprocessors, creates, manages, schedules and execute threads in SIMT Warps (32)
- Threads in a warp starts at the same program address
- Threads have individual programme counter and state
- A block is always split into warps in the same way
- Having divergent control flow will cause the programme to stall

CUDA Memory model

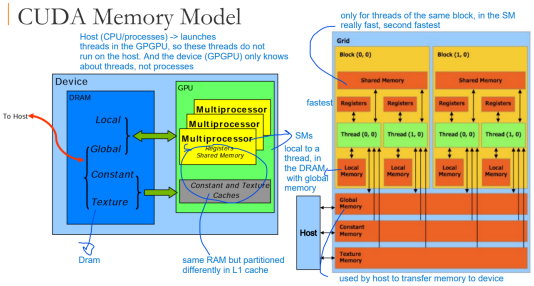


- Kernels are launched in grids
- A block executes on one SM (streaming multiprocessor)
- A block cannot be migrated, but several blocks can reside in one SM
- Register file and shared memory are partitioned among all resident thread blocks

Cuda memory space

- Data must be explicitly transferred from CPU to device
- Shared memory is the cache, and is therefore not cached
- Global, local memory are cached and needs to be warmed up
- Constant memory is useful for uniformly-accessed read-only data
- Spatial data is useful for coherent random-access read-only data (cached too)

Coalesced access

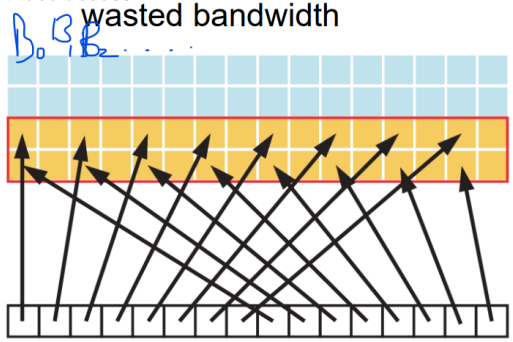


- Simultaneous access to global memory by threads in a warp is coalesced to transactions of 32 bytes
- Reduce disk I/Os

Shared Memory

- Higher bandwidth and lowr latency than local or gloobal
- Divided into equally-sized banks
- Addresses from different banks can be accessed simultaneously
- Bank conflict: two threads access two different addresses in the same memory bank – has to be serialised
- Bank broadcast: (threads accessing the same address in a bank) one reading thread broadcast the result to the conflicting threads so they all get the info

Strided access



- Threads within a warp access memory with a stride size of x
- This increases the number of bank conflicts by x times!
- Half of the elements in the transactions are not used and represent wasted bandwidth

Optimisation in Cuda - goals

1. Maximum memory band width by coalescing memory access
2. Maximise parallel execution by maximising data parallelism and increase hardware utilisation (SIMD!)
3. Maximum instruction throughput by avoiding different execution paths within the same warp

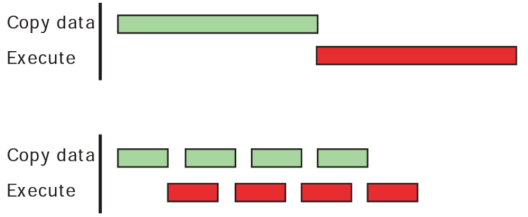
Memory Optimizations

- Minimize data transfer between host and device
- Ensure global memory are coalesced whenever possible
- Minimize global memory accesses by using shared memory
- Minimize bank conflicts in shared memory accesses (e.g. adding padding words between every 32 wordss)

Data transfer between host and device

- Peak bandwidth between device and GPU is higher than between host and device
- Hence data transfer between host and device should be minimized
- E.g. running kernel on GPU without any performance benefits over CPU
- Batch small transfers into one larger transfer
- Use paged-locked or pinned memory transfer (not cached) – eliminates a step in memory transfer
- Page pinned: locked in RAM, cannot be moved to Disk. Both CPU and GPU can access them. Overuse can cause performance issues as it cannot be swapped out of RAM.

Concurrent data transfers and executions



- Overlap asynchronous transfers with computation
- $cudaMemcpyAsync()$ instead of $cudaMemcpy()$, and do CPU computation while data transfers
- Use different streams to acheive concurrent copy and execute

Execution Configuration

- Improve occupancy
  - Number of warps should number of processors
  - So every processor has 1 warp to execute
  - High occupancy hides memory latency and when a block synchronises
- Threads per block should be multiples of warp size
  - If one warp blocks, the other can execute. Better coalesced access
  - Use smaller thread blocks to reduce chances of bank conflict
- Limitation on block size
  - Limited by registers and shared resource
  - The kernel prevents launch if the block allocates more thread resources than available
  - This ensures that at least one block can execute
- Multiple contexts
  - If multiple CUDA apps access the same GPU concurrently, there are likely multiple contexts

Maximise instruction output

- Use single precision floats where possible
- Replace integer division and modulo operations with bitwise operations
- Use signed loop counters

Control Flow

- Reduce divergent warps caused by control flow instructions
- Reduce the number of instructions where possible

L9,10: Distributed-programming models

Performance and Scalability of Parallel Programs

L5: Performance of parallel systems

Two Views

- Response Time (user): duration of a program is reduced (start - end time)
- Throughput (computer manager): more work to be done in the same time (jobs per second)

Performance Factors

1. Programming Model: how well the programmer can code (like good language, API etc)
2. Computational Mode: How well the given program runs in the given architecture
3. Architectural Model: interconnnection network, memory organization, execution mode,sync or async processing

Response time in sequential programs

- Wall-clock time
- Comprise of
  - User CPU time: time CPU spends executing program
    - Know that read and write cycles take different time
  - System CPU time: time CPU spends on system instructions. Depends on OS.
  - Waiting time: IO waiting time and execution of ther programs due to time sharing. Depends on the load of the system.

User CPU time

- $Time_{user}(A) = N_{cycle} * Time_{cycle}$
- N_{cycle}
  - Depends on translation of program statements by the compiler into instructions
  - For a program with n instructions:
  - $N_{cycle} = \sum_{i=1}^{n} CPI_i * n_i(A)$
  - $n_i(A)$: number of times instruction i is executed in program A
  - Depends on architecture of the computer system and compiler
  - CPI_i: average number of cycles per instruction i
- **Refinement with memory access**
- $Time_{cycle}$: Execution time for each instruction, $\frac{1}{clockrate}$
- $Time_{user}(A) = (N_{instr}(A) + N_{mm\_cycle}(A)) * Time_{cycle}$
- $N_{mm\_cycle} = N_{read\_cycle} + N_{write\_cycle}$
- $N_{read\_wite\_cycle} = N_{read\_op} * R_{miss} * N_{miss\_cycles}$
- **Miss rates**
  - Two-level Cache example:

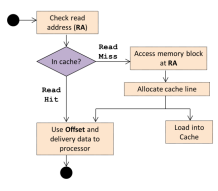
$$T_{read\_access}(A) = T^{L1}_{read\_hit} + R^{L1}_{read\_miss}(A) \times T^{L1}_{read\_miss}$$

$$T^{L1}_{read\_miss}(A) = T^{L2}_{read\_hit} + R^{L2}_{read\_miss}(A) \times T^{L2}_{read\_miss}$$

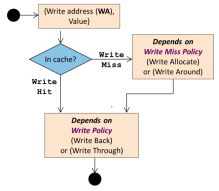
  - Global miss rate: $R^{L1}_{read\_miss}(A) \times R^{L2}_{read\_miss}(A)$

Memory access flow

Read access (load) workflow



Write access (store)



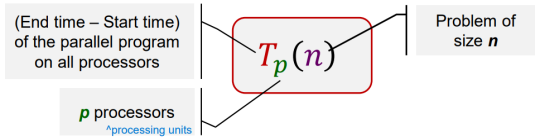
Throughput

- **MIPS** $\frac{N_{instr}}{Time_{user}*10^6}$ OR $\frac{clock_{freq}}{CPI*10^6}$
- Only considers the number of instructions
- Can be easily manipulated by making the instructions smaller so it takes more to run the same programme
- **MFLOPS** $\frac{N_{flops}}{Time_{user}*10^6}$
- Does not differentiate between the different types of floating pt ops

Misc

- Higher clock freq != shorter execution time, since we do not capture CPI

Speed up



- **Cost**: $C_p(n) = p * T_p(n)$, where C_p measures the total work performed by all processors
- A parallel programme is cost optimal if it executes the same total numebr of operations as the fastest sequential program
- **Speed up**: $S_p(n) = \frac{T_{best\_seq(n)}}{T_p(n)}$
- Theoretically $S_p \leq$ p
- Practically, sublinear can occur when the parallel working task fits within the cache but the seq one cannot
- **Efficiency**

$$E_p(n) = \frac{T_*(n)}{C_p(n)} = \frac{S_p(n)}{p} = \frac{T_*(n)}{p \times T_p(n)}$$

speed up / number of cores

- $T_*(n)$ refers to the best sequential
- In the ideal case $T_p = p$, and $E(p)$ = 1

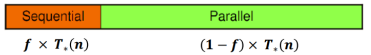
Scalability

- How size of problem and size of parallel computer interact
- Problem size small: Parallelism overheads dominates benefits
- Problem size large: working set cannot fit on machine, cannot start

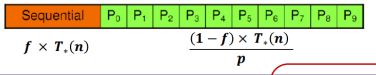
Amadahl's Law

- Speedup of the parallel execution is limited by the sequential fraction $f = \frac{t_{sequential}}{t_{total}}$
- Manufacturers are discouraged from making large parallel computers
- Effort diverted to reducing sequential section

- Sequential execution time:



- Parallel execution time:

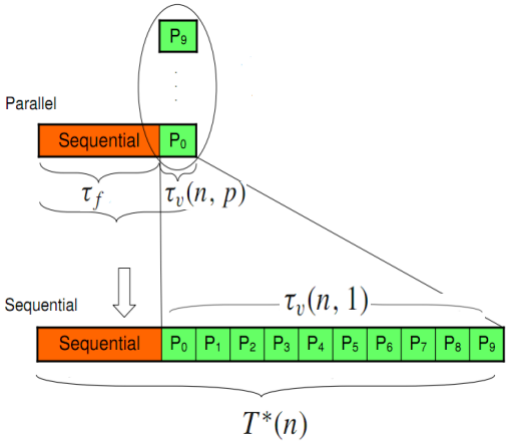


$$S_p(n) = \frac{T_*(n)}{f \times T_*(n) + \frac{1-f}{p} T_*(n)} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

Rebuttal to Amdahl's

- f is not always constant
- In a good parallel programme, $\lim_{n\to\infty}(n) = 0$
- Hence $S_p = p$

Gufstafson's law

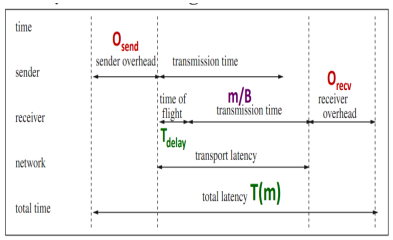


- In some programmes, f decreases when the problem size increases (the parallel parts increases more)
- Then $S_p = p$ given a large enough problem size

Perfromance Measure of Communication

Measure	Definition	Unit
Bandwidth	Maximum rate at which data can be sent	bits (bytes) per second
Byte transfer time	Time to transmit a single byte	Seconds/byte
Time of flight	Time the first bit arrived at the receiver (channel propagation delay)	second
Transmission time	Time to transmit a message	second
Transport latency	Total time to transfer a message = transmission time + time of flight	second
Sender overhead	Time of computing the checksum, appending the header, and executing the routing algorithm	second
Receiver overhead	Time of checksum comparison and generation of an acknowledgment	second
Throughput	Effective bandwidth	bits (bytes) per second

Latency of sending a m sized msg



$T(m) = O_{send} + T_{delay} + m/B + O_{recv} = T_{overhead} + m/B = T_{overhead} + t_B * m$

where B is network bandwidth,
T_{delay} = time first bit to arrive at receiver
no checksum error and network contention and congestion,
$T_{overhead}$ (= O_{send} + T_{delay} + O_{recv}) is independent of the message size;
t_B (=1/B) is the byte transfer time

Finding Possible Bottlenecks

- Instruction-rate limit: Add more non-memory instructions and check if execution time increases linearly with math operations count
- Memory bottleneck: remove most non-memory operations, did the time change proportionately?
- Locality of data access: change all arrays to access A[0]
- Sync overhead: remove all atomic operations or locks (might change control flow, so may not work)

L8: performance instrumentation

New Trends

L12: Energy efficient computing

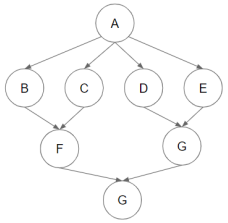
Tutorials

Tutorial 2

Task Dependence Graph

```
A
parbegin
  do
    parbegin
      B
      parallel
      C
    parend
    F
  end
  parallel
  do
    parbegin
      D
      parallel
      E
    parend
    G
  end
parend
H
```

- Fragment 2 Dependencies:



- parallel: X and Y are executed in parallel
- parend: all tasks must complete before moving on

Average CPI

- Not all instructions are created equal! Having fewer instruction != faster programme if translation of instructions takes more clock cycles

Calculaing MIPS

1. Calculate time taken for the programme
   - $\frac{instructions_i * cycles_i}{clockrate}$
   - Ghz = 10^9 Hz
2. Find the total number of million instructions
   - $\frac{totalinst.}{10^6}$
3. divide 2 by 1

Amdahl's vs Gustafson's

- Amdahl's: If problem is fixed sized **OR** there's a constant sequential fraction with increasing problem size, then the speedup is limited by the sequential fraction
- Gustafson's: If the problem size can bve varied **AND** the sequential fraction does not scale much with problem size, then we can solve larger problems with more speed up

Flynn's Taxonomy

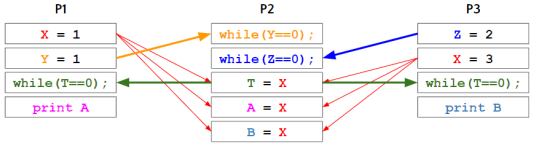
- MIMD is a superset to SIMD!

Tutorial 3

Relaxed Orders

- **Data Dependencies (while loops) DOES NOT COUNT**
- TSO: Relax W $\rightarrow$ R on the same processorf
- PC: Relax W $\rightarrow$ R on the same processor, processes see writes at different times
- PSO: Re;ax WR on the same processor, relax WW on the same processor if no data dependency

Drawing Dependencies



- To determine the final value in a SC execution, draw a line from the last write to the read

Number of Warps and Blocks

- #of registers per kernel=#of registers per thread
- #of blocks=#of threads on device/#of registers per block
- #of warps=#of threads per block/warp size

CUDA mat mul

```
__managed__ float *a;
__managed__ float *b;
__managed__ float *c;


// One thread per output row over some N blocks
__global__ void multiplyKernel() {
    int index = threadIdx.x + blockIdx.x * blockDim.x;

        c[index] = 0;

        for (int j = 0; j < ARRAYSIZE; ++j)
                c[index] += a[index * ARRAYSIZE + j] * b[index];
}
```

```
__managed__ float *a;
__managed__ float *b;
__managed__ float *c;


// One thread per output row over some N blocks
__global__ void multiplyKernel() {
    int index = threadIdx.x + blockIdx.x * blockDim.x;

        c[index] = 0;

        for (int j = 0; j < ARRAYSIZE; ++j)
                c[index] += a[index * ARRAYSIZE + j] * b[index];
}
```

- 1 thread process 1 element in the output array
- Use managed to let CUDA figure out when to copy the matrices

Labs

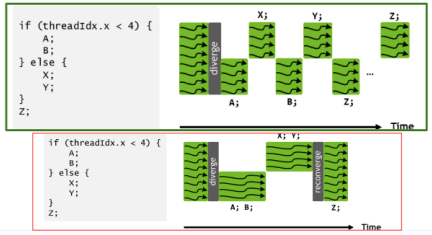
Lab 3: CUDA

Basics

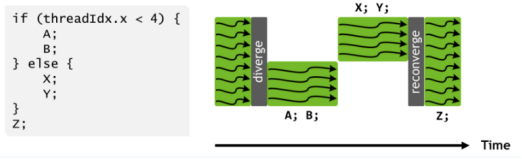
- Complex cores like CPU has low latency
- Many simple cores like GPU has high throughput
- one SM runs one thread block and executes multiple warps of threads in parallel

Volta (CC7) Vs Pascal(CC6)

- Independent thread scheduling (ITS)
- Different statements in the divergent branches can **interleave**



- **Pascal and earlier:** entire divergent region must converge first



CUDA Memory Types

Type	Scope	Access type	Speed	CUDA declaration syntax	Explicit sync
Register	thread	RW	fastest	Compiler decides (from local vars)	no
Local	thread	RW	depends*	float x;	no
Shared	block	RW	fast	__shared__ float x;	yes
Global	program	RW	slow	__device__ float x;	yes
Constant	program	R	slow	__constant__ float x;	yes
Texture	program	R	slow	__texture__ float x;	yes

- program scope = both host and device
- Prefers register local shared global
- **Global memory**
  - $cudaError\_tcudaMalloc(void * *devPtr, size\_tsize)$
  - Visible to all blocks

```
// Malloc host memory
start = (int*)malloc(num_elements * sizeof(int));
// "Malloc" device memory
cudaMalloc((void **)&device_mem, num_elements * sizeof(int));

printf("Incrementor input:\n");
for (i = 0; i < num_elements; i++) {
    start[i] = rand() % 100;
    printf("start[%d] = %d\n", i, start[i]);
}

/**
 * Copy values from start to our CUDA memory
 */
rc = cudaMemcpy(device_mem, start, num_elements * sizeof(int), cudaMemcpyHostToDevice);

if (rc != cudaSuccess)
{
    printf("Could not copy to device. Reason: %s\n", cudaGetErrorString(rc));
}

incrementor<<<1, num_elements>>>(device_mem);
check_cuda_errors();

// Retrieve data from global memory
rc = cudaMemcpy(start, device_mem, num_elements * sizeof(int), cudaMemcpyDeviceToHost);
```

- **Shared memory**
  - $\_\_shared\_\_$
  - Only resides in device, hence faster
  - Only visible to those in the same thread block
- **Unified memory**
  - Defines a common memory addressing space, allowing both CPU and GPU to access it as if it is in their memory space
  - $cudaMallocManaged$ and $\_\_managed\_\_$
  - Page-locked memory (locked in the RAM)! GPU can access directly without CPU intervention.

```
// "Malloc" device memory
cudaMallocManaged((void **)&device_mem, num_elements * sizeof(int));

printf("Incrementor input:\n");
for (i = 0; i < num_elements; i++) {
    device_mem[i] = rand() % 100;
    printf("start[%d] = %d\n", i, device_mem[i]);
}

incrementor<<<1, num_elements>>>(device_mem);
check_cuda_errors();
```

Synchronisation in CUDA

- CUDA provides synchronising primitives
- $atomicAdd(\&counter, 1);$
- **Barrier in CUDA**
  - $\_\_syncthreads()$ synchronises threads in the same block untill all of them have reached this point
  - Threads from other blocks are not synchronised
  - $volatile$ keyword: hints to the compiler to not optimise load and store operations to prevent stale version of the var from being read
  - Volatile varaibles may be modified asynchronously by other threads

Cuda Malloc

- $cudaMalloc(void * *pointer, size\_tnbytes)$ is called in host. Since host cannot touch the shared memory, the memory is allocated to global
- $cudaMemset(void * pointer, intvalue, size\_tcount);$
- $cudaFree(void * pointer)$

CUDA Example Codes

- **Adding two arrays**

CUDA C Program

A CUDA kernel

```
__global__
void addMatrixG( float *a, float *b, float *c, int N )
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   int index = i + j * N;
   if ( i < N && j < N )
      c[index] = a[index] + b[index];
}
```

Device code

```
void main()
{
   ......
   dim3 dimBlk( 16, 16 );
   dim3 dimGrd( N/dimBlk.x, N/dimBlk.y );
   addMatrixG<<<dimGrd, dimBlk>>>( a, b, c, N );
}
```

Host code

```
// Allocate vectors in device memory
size_t size = N * sizeof(float);
float *d_A;
cudaMalloc( (void**)&d_A, size );
float *d_B;
cudaMalloc( (void**)&d_B, size );
float *d_C;
cudaMalloc( (void**)&d_C, size );

// Copy vectors from host memory to device memory
// h_A and h_B are input vectors stored in host memory
cudaMemcpy( d_A, h_A, size, cudaMemcpyHostToDevice );
cudaMemcpy( d_B, h_B, size, cudaMemcpyHostToDevice );

// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) /
                    threadsPerBlock;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

// Copy result from device memory to host memory
// h_C contains the result in host memory
cudaMemcpy( h_C, d_C, size, cudaMemcpyDeviceToHost );

// Free device memory
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

- **Matrix multiplication**

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
   int width;
   int height;
   float* elements;
} Matrix;

// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
   // Load A and B to device memory
   Matrix d_A;
   d_A.width = A.width; d_A.height = A.height;
   size_t size = A.width * A.height * sizeof(float);
   cudaMalloc((void**)&d_A.elements, size);
   cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);

   Matrix d_B;
   d_B.width = B.width; d_B.height = B.height;
   size = B.width * B.height * sizeof(float);
   cudaMalloc((void**)&d_B.elements, size);
   cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);

   // Allocate C in device memory
   Matrix d_C;
   d_C.width = C.width; d_C.height = C.height;
   size = C.width * C.height * sizeof(float);
   cudaMalloc((void**)&d_C.elements, size);

   // Invoke kernel
   dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
   dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
   MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

   // Read C from device memory
   cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);

   // Free device memory
   cudaFree(d_A.elements); cudaFree(d_B.elements); cudaFree(d_C.elements);
}
```

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
   // Each thread computes one element of C
   // by accumulating results into Cvalue
   float Cvalue = 0;

   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;

   for (int e = 0; e < A.width; ++e)
      Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];

   C.elements[row * C.width + col] = Cvalue;
}
```

xs-4114 Vs i7-7700