1 Erlang

```
% boolean expressions
not true.
true and false.
true or true.
true xor true.

length(List). % return the length of List
max(A,B). % return the max
abs(A). % return the absolute value
element(Index, Tuple). % return the element at position Index in the Tuple
[X * X || X <- lists:seq(1,5)]. % [1,4,9,16,25]

% lists
lists:seq(N). % return a list with length N
lists:sum(List). % return the sum of List
lists:map(fun() -> ok end, List). % apply function to each element in the List
lists:nth(N, List). % return Nth element in the List
lists:foldl(fun(Val, Acc) -> ok end, Acc, List).
lists:mapfoldl(fun(X, Sum) -> {2*X, X+Sum} end, 0, [1,2,3,4,5]). % {[2,4,6,8,10],15}

% if-statement
if a == b -> 1;
b == c -> 2;
true -> 3
end.

% switch-statement
case Q of
a -> 1;
b -> 2;
c -> 3
end.

% sending and receiving messages
Pid ! Expr.
receive
Pattern1 -> Expr1;
Pattern2 -> Expr2;
end.
% a new process is created- the child
% when fun returns, the child process terminates, return value is discarded.
spawn(fun()-> ok end)

% bitwise
2#0101 bor 2#1100 -> 2#1101. % base2
16#a -> 10. % base16

Acc ++ List. % O(|Acc|) work for the ++ operation.
```

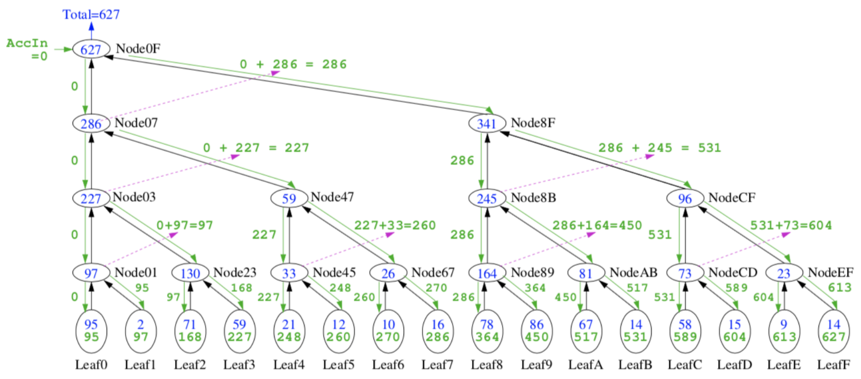
- **Referential Transparency:** a variable get a value when it is declared and that the value of the variable never changes.
- **Tail-recursive:** In comparison to headFactorial, tailFactorial is tail recursive because the recursive call is the very last thing that the function does.
- **Head-recursive:** the recursion call is made while there is still a pending operation for the current call of the function.
- **Tail-call elimination:** overwrite the caller's stack frame with a new frame for caller (tail-call elimination turns tail-recursive functions into while-loops)

2 Reduce

$$\text{Speedup} = \frac{T_{seq}(N)}{T_{seq}(N)/P + \lambda\lceil\log_2(P)\rceil}$$

- A sufficient condition of using reduce is associative.
- **Associative:** $(A \circ B) \circ C = A \circ (B \circ C)$
- **Commutative:** $A \circ B = B \circ A$

3 Scan

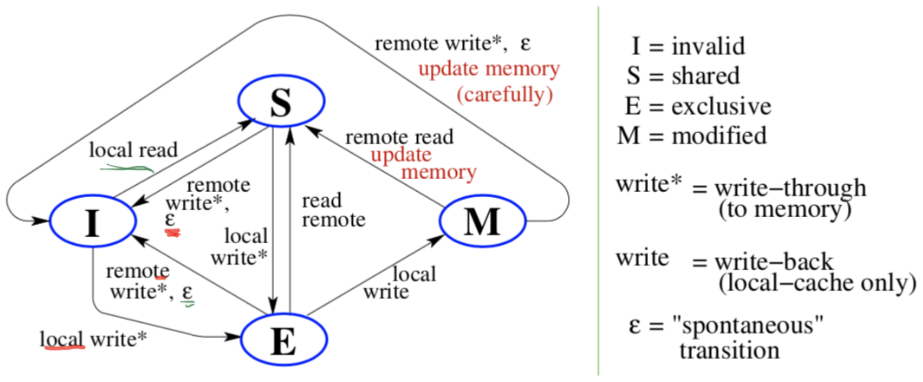


- Scan in two passes
- Upward Pass: Reduce
  - Each node receives values from left and right subtrees.
  - The node combine these values, and send the results to the parent.
  - The combine at the root produces the total for all nodes.
- Downward Pass:
  - Each node has the values from its left and right children.
  - When a node receives a value from its parent:
    - it sends the parent's value to its left subtree.
    - it combine the parent's value with the value from the left subtree and sends this to the right subtree.
  - When a leaf receives a value from its parent
    - It combine this value with its own value and records that as its value as its result

```erlang
sum_scan(W, SrcKey, DstKey) ->
wtree:scan(W,
    fun (ProcState) ->
        lists:sum(workers:get(ProcState, SrcKey)
    end,
    % Leaf1
    fun (ProcState, AccIn) -> % Leaf2
        MyList = workers:get(ProcState, SrcKey),
        {Result, _Total} = lists:mapfoldl(fun (E, Acc) -> V = E + Acc, {V, V} end, AccIn, MyList),
        workers:put(ProcState, DstKey, Result)
    end,
    fun (Left, Right) -> Left + Right end,
    % Combine
    0). % Acc0
```

4 Shared Memory Multiprocessors - Parallel Models

4.1 The MESI protocol



- **Write-back:** writes only update the cache, main memory updated when the cache block is evicted.
- **Write-through:** writes update cache and main-memory.
- Caches can **shared read-only** copies of a cache block.
- When a processor writes a cache block, the **first write** goes to main memory
  - The other caches are notified and invalidate their copies.
  - This ensures that writeable blocks are exclusive.
- MESI guarantees **sequential consistency**.
  - All memory reads and writes from all processors can be arranged into a single, sub-sequential order.
- **Weak Consistency:** Reads can move ahead of writes to maximize program performance.

4.1.1 Implementing MESI: Snooping

- Caches read and write main memory over a shared memory bus.
- Each cache has two copies of the tag: one for the CPU, the other for the bus.
- If the cache sees another CPU reading or writing block that is in this cache, it takes the action specified by the MESI protocol.

4.1.2 Implementing MESI: Directories

- Main memory keeps a copy of the data and
  - a bit-vector that record which processors have copies, and
  - a bit to indicate that one processor has a copy and it may be modified
- A processor accesses main memory as required by the MESI protocol.
  - The memory unit sends messages to the other CPUs to direct them to take actions as needed by the protocol
  - The ordering of these messages ensures that memory stays consistent.

4.1.3 Comparison

- Snooping is simple for machines with a small number of processors.
- Directory methods scale better to large numbers of processors.

5 Message Passing Computers - Parallel Models

5.1 Performance Consideration

- **Bisection bandwidth:** find the worst way to divide the processors into sets of $\frac{P}{2}$ processors each $\times$ the bandwidth.

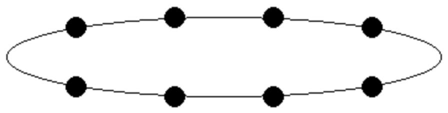
5.2 Network Topologies

5.3 Diameter, Bisection Bandwidth, and Port Number of Each Network

Network	Diameter	Bisection bandwidth	# ports	Latency
Ring Network	$P/2$	2	2	$O(P)$
Star Networks	1	P	P	$O(1)$
ND Meshes	$N \times (\sqrt[N]{P} - 1)$	$\sqrt[N]{P^{N-1}}$	2N	$O(\sqrt{P}), N = 2$
ND Torus	$\frac{N}{2} \times \sqrt[N]{P}$	$2\sqrt[N]{P^{N-1}}$	N/A	$O(\sqrt{P}), N = 2$
ND Hypercubes	$\log_2 P = N$	$\frac{P}{2}$	$\log_2 P = N$	N/A

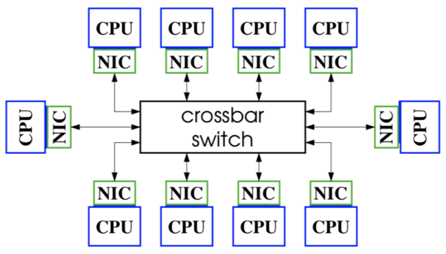
5.3.1 Ring Network

- **Advantages:** Simple
- **Disadvantages:**
  - Worst-case latency grows as $O(P)$
  - Easily congested — limited bandwidth.



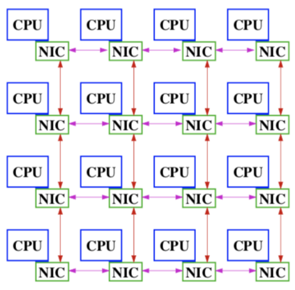
5.3.2 Star Networks

- **Advantages:**
  - **Low-latency:** single hop between any two nodes.
  - **High-bandwidth:** no contention for connections with different sources and destination
- **Disadvantages:**
  - Amount of routing hardware grows as $O(P^2)$.
  - Require lots of wires, to and from switch.

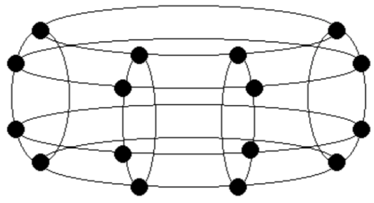


5.3.3 Meshes

- **Advantages:**
  - **Easy to implement:** chips and circuits boards are effectively two-dimensional.
  - Cross-section bandwidth grow as $\sqrt{P}$.
- **Disadvantages:**
  - Worst-case latency grows as $\sqrt{P}$.
  - Edges of mesh are "special cases".



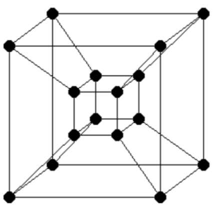
5.3.4 Tori



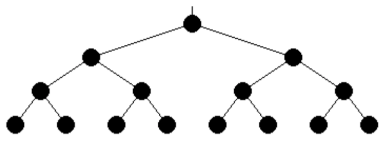
- **Advantages:**
  - Has the good features of a mesh.
  - No special cases at the edges.
  - Interconnect is simpler and takes less space.
  - more volume efficient
  - shorter wires save energy
  - make lower latency between neighbours
  - bounded degree nodes are more modular thus easier to implement
- **Disadvantages:**
  - Worst-case latency grows as $\sqrt{P}$.

5.4 Hypercubes

- **Advantages:**
  - Small diameter $\Rightarrow$ Small worst-case latency
  - high bisection bandwidth $\Rightarrow$ fewer communication bottlenecks.
  - easily space shared
  - communication structure matches divide-and-conquer algorithms such bionic sort
  - more flexible routing avoids congestion
  - No special cases at edges.
- **Disadvantages:**
  - Doesn't become "all wire" for networks with a large number of processors.

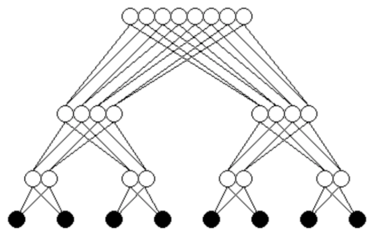


5.4.1 Trees



- **Advantages:**
  - Simple network: # of routing nodes = # of processors - 1
  - Wiring: $O(\log_2 N)$ extra height, $O(N \log_2 N)$ extra area.
  - **Low-latency:** $O(\log_2 N)+$ wire delay.
- **Disadvantages:**
  - **Low-bandwidth:** bottleneck at root.
- **Diameter and Bisection bandwidth**
  - **Binary Tree**: P = 15, Diameter = $2 \times \log_2 \frac{P+1}{2} = 6$, Bisection bandwidth = 1
  - **3-ary Tree**: P = 13, Diameter = 4, Bisection bandwidth = 3
  - **4-ary Tree**: P = 21, Diameter = 4, Bisection bandwidth = 2

5.4.2 Fat Trees



- **Diameter and Bisection bandwidth**
  - **Binary Fat Tree**: P = 8, Diameter = $2 \times \log_2 P = 6$, Bisection bandwidth = P

6 Superscalar Processors: Computer Architecture - Parallel Models

- **Dependencies:**
  - Read-After-Write (RAW): $inst_j$ reads a register or memory location written by $inst_i$
  - Write-After-Read (WAR): $inst_j$ writes a location read by $inst_i$
  - Write-After-Write (WAW): $inst_i$ and $inst_j$ write the same location.
  - Control Dependencies: $inst_i$ is a control flow instruction, we need to execture the correct instructions.
- **Hazards:**
  - **Data Hazards**:
    - an instruction read a different value than would have been read with a sequential execution of instructions
    - if a register or memory location is left holding a different value value than it would have had in a sequential execution
  - **Control Hazards**:
    - an instruction is executed that would not have been executed in a sequential execution.
    - This is because the instruction depends on a jump or branch that hasn't finished in time.
- **Handling Hazards:**
  - Bypass, if an instruction has a result that a later instruction needs, the earlier instruction can provide that result directly without waiting to go through the register file.
  - Move common operations early.
  - If nothing else helps, stall.

6.1 Register Renaming

- **False Dependencies**: WAR: if we had a fresh variable, we could have the write go to a new register and not interface with the earlier (in program order) read. Likewise for WAW.
- When an instruction is decoded:
  - The logical register that it reads are bound to physical registers according to the current register mapping
  - A fresh register is allocated for the register it writes. The register is marked as **busy**.
- When an instruction updates its destination register, it changes it from **busy** to **ready**.
- An instruction can execute when all registers that it reads are ready.

6.2 Branch Prediction / Speculative Execution

- Track statistics for branch outcomes.
- Speculatively execute the more likely path.
- Roll-back if wrong.

6.3 Superscalar Execution

- Fetch several instructions each cycle.
- Decode them in parallel, and send them to issue queues for the appropriate functional unit.
- Handling Dependencies using **Register Renaming** and **Branch Speculation**
- **Multi-threading:** The features for executing multiple instruction in parallel work well for mixing instructions from several threads or processes.

7 Comparisons among Shared Memory, Message Passing, and Superscalar

Whether shared memory or message passing is faster depends on the problem being solved, the quality of the implementations, and the system (s) it is running on. For example, on a single server, it will probably be easier and higher performance to use a shared memory programming environment. Across a distributed cluster, it will probably be faster to use a message passing library.

7.1 Shared Memory

- **Advantages:**
  - **High bandwidth:** the buses that connect the cache can be very wide, especially if the caches are on a single chip.
  - **Low latency:** the hardware handles moving the data – no os calls and context-switch overheads.
  - One thread can pass an entire data structure to another thread just by giving a pointer.
  - No need to pack-up trees, graphs, or other data structure as messages and unpack them at the receiving end.
- **Disadvantages:**
  - Poor scaling of coherence models to large numbers of processors. Shared memory **doesn't scale** as well as message passing.
    - In a message passing machine, each CPU has its own memory, nearby and fast.
    - For large machines, the latency of directory accesses can severely degrade performance.
    - Shared memory moves the data after the cache miss, this stalls a thread.
  - It's easy to overlook synchronization (control to shared data structures). Then we get data races, corrupted data structures, and other hard-to-track–down bugs.
  - A defensive reaction is to wrap every shared reference with a lock, but locks are slow.

7.2 Message Passing

- **Advantages:** reduce the need for synchronization construct.
- **Disadvantages:** Network delays.

7.3 Superscalar

- **Advantages:**
  - Scientific computing
  - Commercial computing (databases, webservers):
    - Have large data set and high cache miss rates
    - find executable instructions after cache miss
- **Disadvantages:**
  - Limited instruction level parallelism.
  - Burning lots of power
  - Many operations require hardware grows quadratically with W

8 Parallel Performance, Speedup and Efficiency

8.1 Parallel Performance, Speedup and Efficiency

- **Latency:** time from starting a task until it completes.
- **Throughput:** the rate at which task are completed.

$$\text{throughput} = \frac{1}{\text{latency}} \qquad \text{sequential programming}$$

$$\text{throughput} \geq \frac{1}{\text{latency}} \qquad \text{parallel programming}$$

- **Speed-up** is a number that measures the relative performance of two systems processing the same problem.

$$\textbf{Speed-up} = \frac{T_{seq}}{T_{par}}$$

- **Efficiency** is a related measure of what fraction of the P processors are kept busy

$$\textbf{Efficiency} = \frac{Speedup}{P}$$

8.2 Amdahl's Law

$$T_{par} = T_{seq} \times (s + \frac{1-s}{P})$$

- Give a sequential program where
  - fraction s of the execution time is inherently sequential.
  - fraction $1 - s$ of the execution time benefits perfectly from speed-up.
- Amdahl's law assumes that the part of the code that can improve from parallel computation achieves perfect speed-up; in other words, that portion of computation achieves a speed-up of P when executed with P processors. In reality, parallel overheads (such as those mentioned above) keep us from achieving such a speed-up.
- **Gustafson's Law:** Many computation have s that decreases as N increases.
- Parallelism offers **modest returns** unless the problem is of fairly low complexity.

8.3 Brent's Lemma

$$T_P \leq \frac{T_1}{P} + T_\infty$$

$$\text{Speed-up}_P = \frac{T_1}{T_P} \geq \frac{T_1}{T_1/P + T_\infty} = \frac{P}{1 + \frac{T_\infty}{T1} P}$$

- T_1 = sequential time = Work
- T_∞ = unlimited parallelism time = Span
- Brent's Lemma provides an upper bound on time and thus a lower bound on speed-up.
- By using the work-span graph, Brent's lemma accounts for computations that have limited parallelism, even if they are not purely sequential.

8.4 Work and Span

- Vertices correspond to operations.
- Edges represent **dependencies**.
- **Work:** is the total number of vertices, corresponding to the sequential execution time.
- **Span:** is the longest path from an initial vertex to a final vertex.
- Work-span model provide the upper-bound and lower bound of speedup.

8.5 Super-Linear Speedup

- **Super-Linear Speedup:** Speedup $>$ P
  - This can occur because the parallel machine has more fast memory (e.g. registers, cache, DRAM) in total than a single processor, and can have a higher fraction of its data references going to faster memory.
  - Another cause is multi-threading where several threads can make better use of the resource of a super-scalar processor than a single thread can.
- **Linear Speedup:** Speedup $=$ P
- **Sub-Linear Speedup:** Speedup $<$ P

8.6 Embarrassingly Parallel Problems

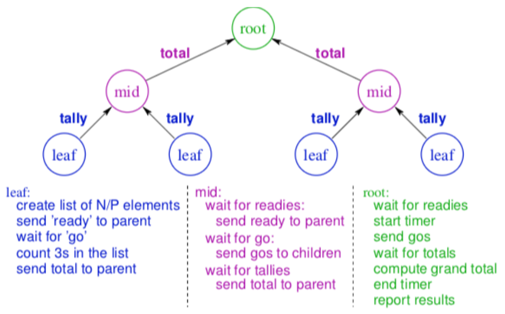
- Problems that can be solved by a large number of processors with very little communication or coordination.
- Overheads which is likely to impact on embarrassingly parallel problems, idle processes, extra computation, and extra memory.

9 Performance Loss

9.1 Overhead

Overhead is the work the parallel code has to do that isn't needed in the sequential program.

- **Communication:** parallel processes need to exchange data. Example: Reduce



  - `shared memory:` The caches communicate to make sure that all references from different cores to the same address look like there is one, common memory. Global memory access overhead is longer than shared memory access.
  - `message passing:` The time to transmit the message through the network. The time set up the transmission and the time to receive the message, etc.
  - `GPU Example:`
    - GPUs communicate using memory, and memory accesses are slow. Threads in the same block can communicate using shared memory, which is fairly fast. Communication between different requires using global memory – either by using atomics or by launching multiple kernels. These global memory accesses introduce communication overhead that can be very large.
    - Communication between CPU and GPU involves slow, global-memory access.

- **Synchronization:** Parallel processes may need to synchronize to guarantee that some operations (e.g. file writes; access to shared data structures) are performed in a particular order. For a sequential program, this ordering is provided by the program itself.
  - `shared memory:` there are explicit locks or other synchronization mechanisms.
  - `message passing:` synchronization is accomplished by communication.
  - `GPU Example:`
    - Calls to `__syncthreads()` incur synchronization overhead.
    - Using multiple kernels to provide communication between blocks. All threads in one kernel must complete execution before the next kernel is launched.

- **Computation Overhead:** A parallel program may perform computation that is not done by the sequential program, e.g. redundant computation: it is faster to recompute the same thing on each processor than to broadcast.
  - `GPU example:` anytime a value is recomputed by multiple threads.

- **Memory Overhead:** Each process may have its own copy of a data structure.
  - `GPU example:` the parallel sieve, each process had its own copy of the first $\sqrt{N}$ primes.

9.2 Limited Parallelism

- **Non-parallelizable code:** something that has to be done sequentially.
- **Idle Processors:** There's work to do, but some processor are waiting for something before they can work on it.
  - Start-up and completion cost.
  - Work imbalance.
  - Communication delays.
  - `GPU example:`
    - Thread divergence, e.g, when only some SPs are active during a reduce.
    - If a SM doesn't have enough warps, then the SM will be idle on cycles that the warp-schedule cannot issue an instruction.

- **Resource Contention:**
  - Too many processors overloading a limited resource.
  - It's easy to change a compute-bound task into an I/O bound one by using parallel programming.
  - Network bandwidth.
  - `GPU example:` Bank conflicts when accessing shared memory.

10 Models of Parallel Computation

10.1 The Parallel Random Access Machine(PRAM) Model

- A computer is composed of multiple processors and a *shared memory.*
- The processor are like those from the RAM model, operate in lockstep.
- The memory allows each processor to perform a read or write in a single step. Multiple reads and writes can be performed in the same cycle.

10.2 The Work-Span Model

$$T_\infty \le T_P \le \frac{1}{P}(T_1 - T_\infty) + T_\infty$$

- **Limitation:** Work-span ignores communication cost.

10.3 The Candidate Type Architecture (CTA) Model

- A computer is composed of multiple processors.
- Each processor has *local memory* that can be accessed in a single processor step.
- A small number of connections to a communication network.
- Sending W words cost $\lambda + t_w W$, λ means communication delay and overhead.
- **Limitation:** Communication is expensive, but it doesn't explicitly charge for bandwidth.

10.4 The LogP Model

- `L`, the latency of the communication network fabric
- `o`, the overhead of a communication action
- `g`, the bandwidth of the communication network
- `P`, the number of processors
- **Limitation:** logP accounts for bandwidth, but doesn't recognize that all bandwidth is not the same.

11 Energy and Parallel Computing

11.1 Moore's Law

- **Moore's Law:** The number of transistors on a chip will double every year from 1965 through 1975.
- The rate has gradually slowed from doubling every year to doubling every 3 or 4 year.

11.2 Energy and Time

- **Deaned Scaling:**
  - Gate delay scales as λ
  - Clock frequency scales as $\frac{1}{\lambda}$
  - Power = Energy / Time
  - Number of devices on a chip scales as λ^{-2}

11.3 The End? of Moore's Law

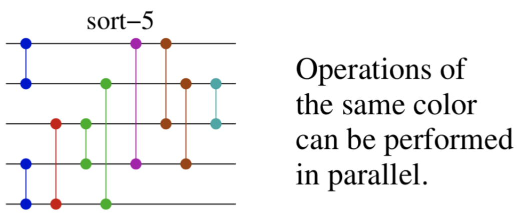
- The power wall — chips are at the cooling limit.
- The atom wall — transistors size are now a few tens of atoms.

11.4 Why Parallelism Matter?

- Greater throughput with a huge number of simpler, lower clock frequency processors.
- The only way go grow performance is with more parallelism.

12 Sorting Networks

- A sorting network is either the identity network or a sorting network, S composes with a **compare-and-swap module** such that two output of S are the inputs to the compare-and-swap, and the output of the compare-and-swap are output of the new sorting network.



- Each *compare-and-swap* is a vertex of the work-flow graph.Z
- Connection between compare-and-swaps are edges in the work-flow graph.
- **Work:** number of compare-and-swap elements.
- **Span:** number of "levels" in the "most-compact" drawing of the network.

12.1 The 0-1 Principle

- If a sorting network correctly sorts all inputs consisting only of 0s and 1s, then it correctly sorts inputs consisting of arbitrary values.
- **Monotonicity Lemma:**
  - Let S be a sorting network with n inputs an N outputs.
  - Let f be any monotonic function, if $x \leq y$, then $f(x) \leq f(y)$.
  - $S \circ f \equiv f \circ S$

12.1.1 Bitonic Merge

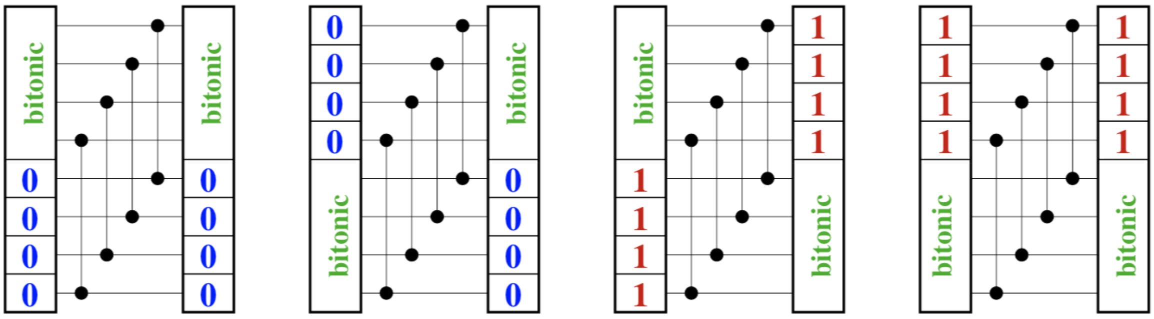
- Recursion assumption: input is two, **sorted** vectors of equal length.
- **Monotonic sequences:**
  - A sequence is **monotonically increasing** if $X_0 \leq X_1 \leq \cdots \leq X_{N-1}$.
  - A sequence is **monotonically decreasing** if $X_0 \geq X_1 \geq \cdots \geq X_{N-1}$.
- **Bitonic Sequences:** A sequence is bitonic if it consists of a monotonically increasing followed by a monotonically decreasing sequence.
  - Either of those sub-sequences can be empty.
  - Any subsequence of a bitonic sequence is bitonic.
- Let X be a monotonically increasing of 0s and 1s of length N.

$$Z_i = \min(X_i, X_{i+\frac{N}{x}}) \qquad 0 \leq i < \frac{N}{2}$$

$$Z_i = \max(X_{i-\frac{N}{2}}, X_i) \qquad \frac{N}{2} \leq i < N$$

12.2 Bitonic Sort

- Divide into two halves of size $\frac{N}{2}$, **Parallel:** sort each half.
- Combine the two, sorted halves into one bitonic sequence of length N.
- Create a clean (a sequence is all 0s or all 1s) half of length $\frac{N}{2}$ and a bitonic half of length $\frac{N}{2}$.

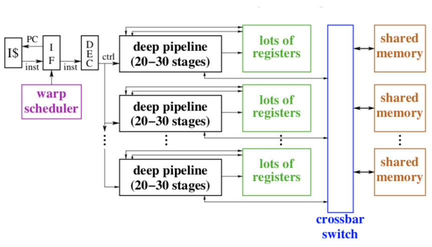


- Recursively merge the two halves, **Parallel:** marge each half.
  - Total parallel time/**Span**: $\log_2 N$
  - Total number of compare-and-swaps/**Work** $\frac{N}{2} \log_2 N$
- **Complexity:**
  - Total Parallel Time/**Span**: $\sum_{k=1} \log_2 Nk = O(\log^2 N)$
  - Total Number of Compare-and-Swaps/**Work**: $\sum_{k=1} \frac{N}{2} \log_2 N = O(N \log^2 N)$

13 Data Parallel Computing and CUDA

13.1 GPUs and Data Parallelism

- **Data Parallelism:** computation that does the same thing to lots of data
- Data-parallel programming often based on "outer-loop" parallelism, leaving the parallelism of the inner loop for instruction level parallelism



- GPUs designed for data-parallel computing, numerical computation.
- GPUs are better than CPUs because they amortized the overhead of instruction fetch, decode, and pipeline control across many execution units.
- GPUs have higher main-memory bandwidth than CPUs because GDDR is faster than DDR.
- **CPUs** benefit from multiple threads so that one thread can execute while another is blocked due to a cache miss.
- GPUs are **Single-Instruction, Multiple-Data(SIMD)** machines
  - Each instruction is executed for many data streams using many pipelines to hide latency
  - Deep pipelines: breaking instruction **execution** into small steps allows simple hardware to get good performance. More time per operation means less energy, latency is high, the throughout remains one instruction per cycle per pipeline
  - This amortized the cost of instruction fetch, decode, and control
  - The lock-step execution of the pipelines simplifies synchronization issues
  - No bypasses, With so many pipeline stages, bypassing become impractical. Each instruction must go all the way through the pipeline before another instruction can use the results.
- Memory
  - Memory accesses are a major bottleneck
  - Caches are poor choice: one miss stalls all threads in a warp, so use **shared memory**
- Energy
  - x86: energy goes to instruction fetch, decode and other control issues, energy for ALU operations is negligible.
  - GPUs: Register files are large, register file read and writes dominate the energy budget, about 10× more energy efficient than the x86, energy for ALU operations is negligible.
- Why GPU need thousands of active threads to fully utilize the processors?
  - `Mitigating data hazards:` Using lots of threads allows the SPs to have deep pipelines(easier to implement, lower power, higher clock speeds) without the complications of bypasses.
    - ∗ A pipeline bypass is a mechanism that allows a result that is in the pipeline from an earlier issued instruction to be made available to a later issued instruction, even if the first instruction is still in the pipeline. Bypasses are energy hogs.
  - `Mitigating control hazards:` Even though a branch instruction takes 10s of cycles on a GPU, the SM can fetch from other warps until the branch is resolved.
  - `Hiding global memory latency:` Accesses to global memory are slow. Other warps can execute while a warp is waiting for a load to finish.
  - `Warps have lots of threads:` This amortizes the hardware, and especially the power consumption for instruction fetch, decode, and other pipeline control issues across many SPs.
  - `A GPU has lots of SMs:` This is how it gets lots of parallelism. But many SMs times many warps per SM times many threads per warp results in needing thousands of threads to get high performance from GPU.

13.1.1 Grids, Blocks, Threads and Warps

- A **grid** is organized as an array of blocks.
- Each **block** is an array of threads, a block must have all execution resources it needs before it is launched
  - A block runs on a single SM.
- `blockDim` gives the number of threads in a block, in the particular direction
- `gridDim` gives the number of blocks in a grid, in the particular direction
- `blockIdx` gives the indices of the thread's block within the grid
- `threadIdx` gives the indices of the thread within its block
- A group of thread that execute together, on each pipeline of a SIMD core are called a **warp**.
- The `warp scheduler` determines which instruction to dispatch next instruction.
  - If the dependencies for the next instruction are resolved, it can execute for all thread of the warp.
  - The hardware in each streaming multiprocessor dispatches an instruction each block cycle if a ready instruction is available.

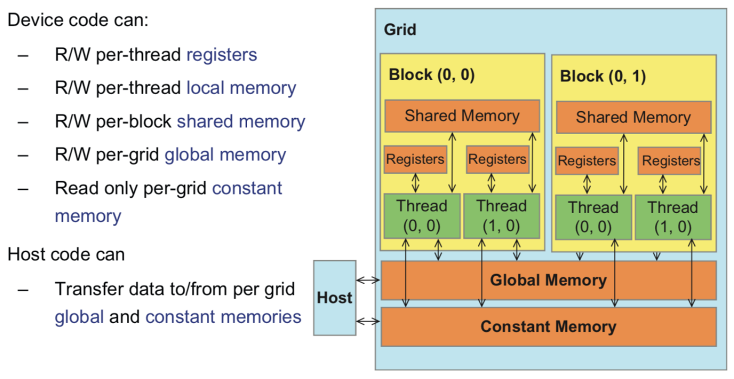
13.2 CUDA

- A CUDA program consists of three kinds of functions:
  - `__host__` **functions:** callable from code running on the host not the GPU, execute on the host CPU
  - `__device__` **functions:** callable from running on the GPU, but not the host, execute on the GPU
  - `__global__` **functions:** called by code running on the host CPU, execute on the GPU
- Execution Model: Memory
  - **Host Memory:** DRAM and the CPU's caches, accessible to host CPU but not to GPU.
  - **Device Memory:** GDDR DRAM on the graphics card, accessible by GPU, the host can initiate transfer between host memory and device memory. *BUT* host cannot read/write the device memory directly.
- One or more host functions, including `main` to run on the host CPU.
  - Allocate device memory.
  - Copy data from host memory to device memory.
  - Launch the device kernel by calling `__global__` function.
    * Data parallel code that runs on the GPU is called a **kernel**.
    * Invoking a GPU kernel is called **launching** the kernel.

```
int main(void) {
    ...
    float a = 3.0;
    saxpy<<<ceil(n/256.0),256>>>(n, a, dev x, dev y); // create n / 256 blocks, each blocks have 256 threads
    cudaMemcpy(yy, dev y, size, cudaMemcpyDeviceToHost);
    ...
}
```

  - Copy the result from device memory to host memory.
- `__syncthread()`: all the threads in the block must execute this statement before any continue beyond it, all threads in the block must meet at the barrier and they must meet at the same barrier.
- To get good performance:
  - perform many operations for each value copied between memories
  - perform many operations in the GPU fro each access to global memory
  - enough thread to keep the GPU cores busy

13.3 CUDA Memory



Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

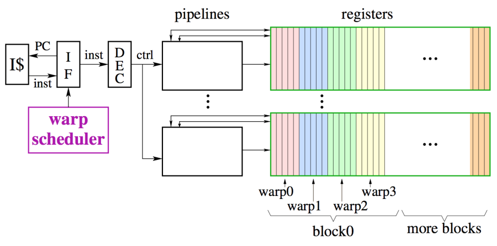
Host code can

- Transfer data to/from per grid global and constant memories

FIGURE 4.6

Overview of the CUDA device memory model.

- **Registers**



  - Each SP(pipeline) has its own register file
  - The register file is partitioned between threads executing on the SP (32 4-byte registers per thread)
  - Local variables are placed in registers
  - in recent CUDA, threads in the same warp can swap registers
  - Performance trade-off:
    - ∗ A thread can avoid slow, global memory accesses by keeping data in registers
    - ∗ But using too much registers reduces the number of threads that can run at the same time.
    - ∗ In general, the more resources each thread requires, the fewer the threads that can reside in each SM, and likewise, the fewer the threads that can run in parallel in the entire device.
    - ∗ if a thread use more registers, register will spill to main memory

- **Shared Memory**
  - On-chip, the shared memory for each SM is partitioned into **banks**
  - allocated to a block, all threads in a block access the same memory
  - Shared memory is a limited resource (48 to 96 KBytes/SM, whereas Registers 256 Kbytes/SM)
  - There is a crossbar controlling the access, one thread accesses one bank at a time (90% true)
  - For the GPUs we have used, there are 32 banks, selected by bits 3 through 7 of the address. This means that consecutive floats are stored in different banks. More generally, if two floats have indices that are different modulo 32, they will be in different banks.
  - A **bank collision** occurs if two or more threads access **different locations** belonging to the same bank and threads need to access the bank one by one.
  - BUT if multiple threads accesses the **same location** in the same bank, there is **no collision** because threads in the same warp can swap register, directly copy from the first thread
  - **CON:** In general, bank conflicts are undesirable because they cause the code to run slower. References to different banks can be handled in the same clock cycle, but references to different locations in the same bank must be handled on sequentially clock cycles. Thus, conflicts increase the number of clock cycles needed to complete a load or store operation.

- **Global Memory**
  - Off-chip DRAM

    * when we read a location, we get 1 bit from each tile that is accessed, but we had 1024 bits available from each tile => use threads to access it coalesced
  - The memory is mounted on DIMMs. A DIMM typically has 16 / 18 chips
  - Each chip consists of many "tiles", a typical chip have 4096 tiles
  - Each tile is an array of capacitors, a typical tile could have 1024 $\times$ 1024 capacitors
  - Each capacitor holds 1 bit
  - Writing: easy, Reading: hard
  - Terrible latency, fairly high bandwidth
  - **CGMA** stands for "Compute to Global Memory Access ratio"

$$\text{CGMA} = \frac{\#\text{FloatingPointOperations}}{\#\text{GlobalMemoryAccess}}$$

  - CGMA for GPU: Global memory bandwidth can easily be a performance bottleneck for GPU computations. To fully utilize the computing capabilities of a GPU, we need algorithms that perform a fairly large number of operations on each data value read from or written to the global memory. The CGMA describes this ratio of the amount of computation to the number of memory references. In general, higher values for CGMA are better as code with a higher value will usually have less performance loss from memory bandwidth constraints.
  - CGMA for CPU: CPUs can execute a hundred instructions or more in the time that it takes to perform on DRAM access. For CPUs, the usual solution is caches. To get good performance, the average line loaded into a cache must be accessed a large number of times before it is replaced by another line from the DRAM.

- Other Memory
  - Constant memory: caches, read-only access of global memory
  - Texture memory: global memory with special access operations
  - L1 and L2 caches: only for memory reads
- Summary
  - GPUs can have thousands of execution units, but only a few off-chip memory interfaces
    * which means the CPU can perform 10-50 floating point operations per every memory read or write
    * Arithmetic operations are very cheap compared with memory operations
  - Use Registers and the per-block shared memory to mitigate the off-chip memory bottleneck
  - Moving data between different kinds of storage is the programmer's responsibility

13.4 CUDA Performance Considerations

- Floating Point Foibles
  - GPUs are optimized for single precision floating point arithmetic
  - if one operands is double precision, the compilations done using double precision arithmetic
  - A **fused multiply-add**, a common fracture in hardware for floating-point arithmetic where a multiply followed by an add, e.g.,$a * X + b$, can be computed as a single operation. But it counts as 2 floating point operations.
- Shared Memory Accesses faster than global memory, but watch out shared-memory bank conflicts
- Global Memory Accesses
  - **Coalescing references:** If the threads of a warp access consecutive locations of the global memory, the memory reference is said to be coalesced. Coalesced memory references only make one global memory access.
  - **PRO:** The GPU can take advantage of accessing a large, contiguous block of memory and achieve high bandwidth with the data transfer. Conversely, if locations are not coalesced, then several bank accesses may be needed, and several transfers of data from the DRAM to the GPU. Thus, coalesced references are handled faster than non-coalesced ones.
  - coalesced accesses to global memory are faster than worst-case bank collision with the shared memory
- SMs and Thread Occupancy

$$\text{threadOccupancy} \leq \min(8, \lfloor\frac{2048}{\text{threadsPerBlock}}\rfloor) \times \frac{2048}{\text{threadsPerBlock}}$$

  - An SM has warps of 32 threads.

  - An SM can simultaneously execute up to 2048 threads (64 warps).
  - An SM can simultaneously execute up to min(8, $\lfloor 2048/\text{threadsPerBlock} \rfloor$) blocks.
  - An SM has 96K bytes of shared memory.
  - An SM has 64K (2^{16}) 32-bit registers (1K registers/SP).
  - Each block can have up to 1024 threads, we want at least **2** blocks per SM, more is better.
  - Each block can have up to 48K bytes of shared memory.

- Instruction Mix, the program does other operations as well, so optimizing performance can involve minimizing this overhead
  - Good algorithm design
  - Memory access optimization
  - Bigger Kernels: when `do something` is big, kernel launch takes most the time.
  - Loop Unrolling: two or three instruction per loop iteration to manage the loop, so each loop iteration perform multiple copies of the loop body

- **Thread divergence** occurs when different threads in the same warp follow different execution paths. For example, one executes the "then-"clause" of an if-statement and the other executes the "else".
  - **CON:** Thread divergence is undesirable because the GPU must execute both paths, and only some of the pipelines are active for each path. This results in idle execution pipelines.
  - It can be worse,
    * Nested if-then-else statements, divide threads into more than two groups
    * For-loops with thread-dependent bounds, while-loops
  - It can be better,
    * If all threads in a warp are then-threads, then the else-instructions aren't fetched, vice verse.
  - `__syncthreads()` implements a barrier: all threads in the block must reach the barrier before any continue beyond it.
    * e.g. `__syncthreads()` in a then-clause for one thread can't match a `__syncthreads()` in a else-clause for another thread. **deadlock:** the block hangs forever at the `__syncthreads()`
    * If there is a `__syncthreads()` in the body of a for-loop, all threads must reach it on the same iteration.