CS3210 - Parallel Computing
AY25/26, Y3S1
Notes

Sim Ray En Ryan

December 3, 2025

Contents

1 Introduction

1.1 Admin

Cristina Carbunaru, cristina@nus.edu.sg

- 6% 3 Labs, 2% each
- 5% Tutorial Attendance and Participation
- 30% 3 Programming Assignments
- 9% 3 quizzes in Week 5, 8, and 12
- 50% Final Exam

Odd weeks are labs, even weeks are tutorials.

1.2 Motivation

In 2004, Intel hit an inflection point of the Power Density Wall, as they could not increase computing power of a single core anymore.

No matter how fast access and arithmetic are, parallel may be faster. Also good when digital speeds reach the limit, or if for example predicting tomorrow's weather requires more than one day to compute.

1.3 Difficulties

Programming techniques need to change from serial machines, otherwise most resources will sit idle. Traditionally, a problem is divided into a discrete series of instructions, and will be executed by the processing unit (PU), Memory, Control Scheme, memory.

1.4 Ways to improve serial performance

- Work hard: Increasing clock frequency to carry out more instructions per second
- Work smart: Pipelining, superscalar processor (processor with multiple PU)
- Get help: Replication with multicore or cluster

Parallel Computing is the simultaneous use of multiple PUs to solve a problem fast or solve a larger problem.

1.5 Computational Model Attributes

1.5.1 Operation Mechanism

The primitive units of computation and basic actions of computers: Data types and operations of IS

1.5.2 Data Mechanism

How data is accessed and stored in address spaces. Computers now can have very fast speed (El Capitan is 1742 petaflops, but the issue is that people are unable to utilize them properly). GREEN500 ranks how efficient a Supercomputer is.

1.5.3 Control Mechanism

Rules to schedule the units of computation to Primitive Units

1.5.4 Communication Mechanism

Communication among processing elemenets working in parallel to exchange needed information.

1.5.5 Synchronization Mechanism

Ensuring information arrives at the right time.

1.6 Parallel Computing

With a problem, decompose it into tasks, then schedule and map these taks to physical cores and processors.

1.6.1 Decomposition

Multiple decompositions are possible. Granularity decides the size of tasks. Difficult to automate as different 'tasks' have dependencies and need to coordinate with each other.

1.6.2 Tasks

Scheduling: Assignment of tasks to processes or threads. This dictates the execution order, and may be manually scheduled, static, or dynamic

Mapping: Assignment of process and thread to physical cores and processors.

Dependencies among tasks impose constraints on scheduling. To execute correctly, processes and threads need to synchronize and coordinate. (Threads use shared memory and processes use distributed memory)

1.7 Concurrency and Parallelism

Concurrency may not mean processes are running at the same instance, but two or more flows are making progress at the same time due to interleaving or by executing instructions at the same time.

Parallelism happens when two or more tasks execute at the exact at the same time together.

1.8 First Parallel Solution

Summing n numbers together. With p cores, you can calculate the sum of $\frac{n}{p}$ values on each core. Each core now has their own partial sum. Using one core, calculate the global sum.

1.8.1 A better Parallel Solution

Share the work of global summing across the cores too (kind of like merge sort). The improvement factor is much better.

1.8.2 Tradeoffs

Parallel execution time is computation time and parallelization overheads.

Parallelization incurs overhead of

- Scheduling the tasks to processor
- Information exchange and synchronization
- Idle Time

Furthermore

- Finding enough parallelism (Amdahl's Law)
- Granularity
- Locality
- Load Balance
- Coordination and Syncrhonization
- Debugging
- Performance modelling and monitoring

In CS3210, we will learn how to write parallel code for:

- Single and Multicore Processors
- GPUs
- Distributed Systems

and teach the hardware details that affect parallelism, as well as how to measure and optimize the performance (speed, efficiency) of parallel programs.

2 Processes, Threads, and Synchronization

2.1 Parallelization Steps

Decomposition of algorithm into tasks is done by the programmer. Then, scheduling each task to a process and then mapping them to the processors is done by the OS and libraries.

Process an instance of a program in execution. Identitfied by PID, has a PC. global data, OS resources, stack and/or heap, GPRs, address space.

When there are more programs than cores, multi-programming needed. In multi-programming, context switches is needed to switch between processes. Overhead is incured when creating new processes, and saving suspended process states.

Creating a process is costly as there is large overhead of allocating, initializing, copying data. There is also overhead in context switching, as well as communicating between processes through the OS

Execution can either be time slicing (pseudo parallelism) or with execution on different resources.

In Unix, the fork() system call duplicates a process (with same PC, diff PID, diff address space). (Note that to save on memory, the OS will only duplicate the address when one of it writes to it.) 'int exec(char *prog, char *argv[])' can be used to create a 'new' process with different statements. fork() returns the PID of the child, or 0 if it is the child. Fork is useful as it can cooperate with the parent and relies on the parent's data to accomplish its task

Process State Graph

- New
- Ready
- Waiting (Blocked)
- Running
- Terminated: Through exit and wait

IPC

- Shared memory (Need to protect access with locks)
- Mesasge passing (Can be blocking or not blocking, synchronous or asynchronous
  - Exceptions are synchronous, occurs due to execution of program
  - Interrupts are asynchronous, occur independelty of execution. Both are handled by handlers
- Pipes and Signals

2.2 Threads

Have independent control flows, and defines a sequential execution stream. Threads are shared-memory architecture. Different threads can be run on different cores

2.2.1 User Level Threads

Managed by thread library, and the OS is unaware of these threads. Since no context switching is required, no kernel access is required and thus is faster to switch. However, the OS cannot map different threads of the same process to different resources. OS also cannot switch thread if one thread blocks.

2.2.2 Kernel / Hybrid Threads

OS is aware of threads. Many to one mapping has User threads are mapped to one kernel thread, and the thread library is responsible for scheduling the user threads. One to one mapping is where each user level thread is mapped to one kernel thread. Thus no library level scheduler (that is less optimized) is needed.

In many to many mapping, the library scheduler assigns user threads to a set of kernel threads (usually the amount of cores on the machine).

2.2.3 Sharing of data

Threads share text, data, and heap, but each have their own stack frame. If the thread is not active, it will not have a stack. The number of threads should be suitable to parallelism degree of application and available resources. Not too large to reduce overhead and to keep termination small.

2.3 Synchronization

If there is a shared variable, and the variable is read and modified by two or more threads, then access must be controlled to avoid erroneous behavior.

Mechanisms include locks, mutexes, semaphores, monitors, condition variables, etc. Patterns used to coordinate access include bounded buffers and porducer-consumers.

2.3.1 Race Conditions

When two threads are interleaved in just the correct way, race conditions occur. (When multiple execution paths of different threads of processes execute at the same time finish in a different order than exepcted). A critical race condition causes invalid execution and software bugs. Then, critical selections are used to make the operations upon shared states mutually exclusive.

A critical selection cannot be entered by more than one process or thread at a time, and others are suspended or waited until the first leaves.

Locks provide minimal semantics and are used to build the mechanisms below. They have an acquire and release, and are usually a spinlock (busy waiting in CS2106) or a mutex. Mutexes require OS help to block processes.

Semaphores are basic but hard to program with. Semaphores are datatypes with an integers and have a wait and release. There are mutex (binary) semaphores that represent single access, as well as counting semaphores that allow multiple threads to pass throguh the semaphore. A barrier is an implementation of semaphores (refer to CS2106)

Monitors are high level and require language support, have implicit operations

Messages are more straightforward and will be discussed later in the semester.

2.3.2 Deadlocks

Deadlocks exist if among a set of processes if every process waits for an event caused by another process. Occurs when processes compete for access to limited resources and are incorrectly synchronized. More specifically, when there is mutual exclusion, when a process holds a resources while waiting for another, when there is no pre-emption, and there is circular wait.

Livelocks also occur as a result of anti-deadlock mechanisms, when processes constantly change with reagrd to another without any progressing. It is not detectable as to the OS, it seems like both are doing work.

Starvation occurs when a process is prevented from making progress because of other processes holding resources it requires. For OS, a high priority process will always prevent a low priority process from running on the PU.

2.3.3 Synchronization Problems

Producer Consumer The producer waits for the event, enters CS to add an event, then signialling that an item has been added. The consumer waits for an item, then enters CS

to get the event, then processes the event. Look at order of operations carefully to reduce possible number of context switches.

Reader Writer Any number of reader can read, only one writer can be used. For writers:

- roomEmpty.wait()
- Write stuff
- roomEmpty.signal()

And for readers:

- mutex.wait()
- readers += 1
- if readers == 1, roomEmpty.wait()
- mutex.signal()
- Read stuff
- mutex.wait()
- readers -= 1
- if readers == 0, roomeEmpty.signal()
- mutex.signal()

A turnstile can be added (a wait signal in front of readers) to only let them execute when there is no writer.

Can cause starvation of writers if readers never go below 1. Writers can be given priority by waiting on a write switch lock. This means that all current readers will have to finish reading, then the writer gets to finish their task, then continue.

3 Parallel Computing Architectures

Parallelism is the source of perfomance gain, and occurs at bit level, instruction level, thread level, and processor level

3.1 Processor Architecture and Technology Trends

3.1.1 Bit Level Parallelism

Increase processor word size. This increases unit of transfer between processor and memory, integer size, floating point precision, etc. As computers increased in power, words increased from 16 bits to 64 bits. In this sense, more data is being worked on at the same time. For example, a 16 bit computer needs at least 4 times as many operations to add a 64 bit numbers as opposed to a 64 bit computer.

3.1.2 Instruction level Parallelism

Executing instructions in parallel either through pipelining (parallelism across time) or superscaling (parallelism across space).

Pipelining speeds it up by a factor of at most the number of pipelining stages. It however introduces independence, bubbles (delaying), hazards (data and control flow), and additional mechanisms such as prediction and out of order execution are required.

Superscalar duplicates the pipelines, and allows multiple instructions to pass through the same stage. (By putting multiple hardware components at each stage). This causes even more structural hazards (rip CS2100), but also allows for more instructions per cycle.

SIMD Single instruction multiple decode. By duplicating only the ALU, and then sending the same data to multiple ALUS allowing for faster operations on larger datasets.

3.1.3 Thread level Parallelism

ILP is limited as only 2-3 instructions can be executed in parallel due to data and control dependencies. However, threads will have less and thus should be exeucted in parallel. Simultaneous Multi Threading (SMT) can run one scalar instruction per clock from one hardware threads and has two logical cores. Hyperthreading (Intel's implementation) is when each core can execute two threads at a time.

3.1.4 Processor Level Parallelism

Adding more cores to the processor allows each process to be mapped to multiple cores and thus speed up in the most straightforward way.

3.2 Flynn's Parallel Architecture Taxonomy

3.2.1 Single Instruction Single Data SISD

A single instruction stream is executed, and each instruction works on a single data.

3.2.2 Single Instruction Multiple Data SIMD

A single stream of instruction and each instruction works on multiple parts of the data, exploiting data parallelism (Vector Processors), where an operation works on multiple words at the same time.

3.2.3 Multiple Instruction Single Data MISD

Multiple instruction streams, and all instructions work on the same data. Not implemented except in systolic array, as well as when used for reliability checking.

3.2.4 Multiple Instruction MUltiple Data MIMD

Each PU fetches its own instructions, and operated on its own data.

In today, a variant is usually used. nVIDIA GPUs have a set of threads executing the same code (SIMD) and multiple threads executing parallel (MIMD)

3.3 Architecture of Multicore Processors

3.3.1 Hierarchical

Multiple cores share multiple caches. Cache size increases from leaves to root. Each core usually has its own L1 cache and shares L2 cache with other cores. Usually used with standard desktops, server processors, and GPUs.

Smaller cores usually share more cache, and are used to do time-sensitive operations in order to take advantage of parallelism more.

3.3.2 Pipelined

Data elements are processed by multiple execution cores in a pipeliend way. Is useful if the same computation steps have to be applied to a long sequence fo data elements. For example, in routers where they have to unpack and check packet headers and process, or in GPUs and performing matrix operations.

3.3.3 Network Based

Cores and local caches and memories are connected via interconnection network.

3.4 Memory Organization

Processors are more efficient when memory is resident in cache. The memory latency is the amount of time for a memory request (load, store), and a memory bandwidth is the rate at which the memory system can provide data to a processor. A processor stalls when it cannot run the next instruction due to the dependency on a previous instruction. In modern computing, memory is usually the bottleneck.

Thus, we should try to make programs access memory as infrequently as possible to utilize the processor more efficiently. This can be done through exploiting temporal and spatial locality, sharing data acorss threads, or using more operations as opposed to storing and reloading values (so just compute them again!)

3.4.1 Distributed-Memory Systems

Each node is an independent unit with their own processor, memory, and maybe peripheral elements. Memory in each node is private (and physically distributed). Try not to access memory (may be impossible) of other nodes as it will take very long.

3.4.2 Shared-memory Systems

Programs access memory through shared memory provider. The program does not need to know about the actual hardware memory architecture.

Cache coherence Since multiple copies of the same data may exist on different caches, a local update by a PU should not see unchanged data. Thus, we need to make sure that the other caches are changed too (to make it coherent). This causes at least one write operations and triggers at least one read operations.

Uniform MA and Non UMA When the delay to memory is uniform, it is UMA. It is suitable for a small number of PUs.

Cache Coherent and NCC A precense of a local cache with cache coherence protocol. When the same shared variable is in multiple caches, hardware ensures the correctness via a protocol.

4 Parallel Programming Models I

Parallelism is limited by program dependencies (data and control), as well as runtime (memory contention, communication overheads, process overheads, coordination overheads)

Work = Time taken to do actual work + Overhead of Dependencies

4.1 Parallelism and Types of Parallelism

```
for (int i = 0; i < N; i++) {
  a[i] = b[i] * c[i]
  d[i] = b[i] / c[i]
}
```

4.1.1 Data parallelism

Partition the data used in solving the problem among PU. Each PU carries similar operation on its part of the data. From the code snipped above, each PU will operate on $a < i < b$ based on how many PUs there are. Since these iterations are independent of each other

SIMD computers are designed to exploit data parallelism.

Loop parallelism is a type of data parallelism where if iterations of a loop can be executed in an arbitrary order, then it can be executed in parallel on different cores

For example, the pragma operation in C++ can be used to calculate matrix multiplication (scalar product) in parallel.

4.1.2 Task parallelism

Partition the tasks among PUs. In the code snippet above, each PU has a specific task, where one performs multiplication and one performs divisions. Again, they are independent so it is fine.

A task can be a single statement, a series of statements, a loop, or a function call. A single task can then be executed sequentially by one PU, or in parallel by multiple PUs. Tasks must be decomposed properly to have the largest speedup.

A DAG can be used to represent each task, with its expected execution time and edges to represent dependencies. This allows us to find the critical path (worst case execution time) as well as the degree of concurrency that is possible with this task set.

4.1.3 Data or Task Parallelism Example

If there are 60 assignment scripts, each with 15 questions and to be distributed to 3 TAs for marking, is it better to assign 20 scripts to each TA, or 5 questions to each TA. In this case, task parallelism (5 questions per TA for all 60 scripts) is probably better as each TA can mark each question faster by being more familiar with the question.

4.2 Models of Coordination

4.2.1 Shared Address Space

Tasks communicate by reading and writing to shared variables. Mutual exclusion ensured via locks. This requires hardware support to implement efficiently.

4.2.2 Data parallel

Programs perform the same function n different data elements in a collection

4.2.3 Message Passing

Tasks operate within their own private address sapce, and communicate explicitly by sending and receiving messages. This is used in clusters and supercomputers.

It is important to implement abstractions on hardware as message sending requires copying a lot of data from the message.

4.3 Program Parallelization

Transforming sequential computation to parallel computation. The parallelization can be as granular as a sequence of instructions to as corase as a function or method. Follow Foster's design methodology

4.3.1 Partitioning

First partition a problem into many smaller tasks. Divide the computation and data into independent pieces.

Data Centric Divide data into pieces that are approximately the same size. Then decide how to associate computations with data

Computation Centric Divide computation into pieces, then decide how to associate data with computations.

There should be at least 10x more primitive tasks than cores in the program, and they should be roughly the same size. This is to reduce idle time when certain cores finish faster than others. Should also minimize redundant computations and data storage.

4.3.2 Communication

Provide data required by the partitioned tasks (cost of parallelism) (How to build the tasks to get back the main problem). Tasks usually do not all execute independently.

Local Communication Tasks need data from a small number of other tasks. Create channels

Global Communication A significant number of tasks contribute data for computation: Dont create channels early

4.3.3 Agglomeration

Combining tasks into larger tasks. Since the number of tasks is more than the number of cores, we improve performance of cost of task creation and communication. Combining nodes together? TODO. By combining tasks, we make it less granular but also decrease

the cost of communication overhead. To make this better, exploit locality, and make each group of tasks suitable for the target system that it is executing on.

4.3.4 Mapping

Map tasks to processor to reduce total execution time. We want to maximize processor utilization whole also minimizing inter processor communication. The optimal mapping is NP hard. If dynamic task allocation is used, the task allocator cannot bottleneck performance. If static task allocation is chosen, the ratio of tasks to cores is at least 10 to 1.

4.3.5 Automatic Parallelization

It is hard to analyze dependencies autoamtically, and execution time is also difficult to predict at compile time. Thus, usually it is better to parallelize the program yourself.

4.4 Parallel Programming Patterns

Provides cooridnation structure for tasks. For example, Fork Join, Master Worker, Parbegin Parend, SPMD and SIMD, Task Pool, etc

4.4.1 Fork Join

Task T creates child tasks. Each child executes in parallel, can excute the same or different program parts, and join the parent at different times.

4.4.2 Parbegin Parend

Programmer specifies sequence of statements to be executed by set of cores in parallel. It is fork join but when all forks are done adn join at the same time.

4.4.3 Master Worker

A single program controls execution of worker threads. Master responsible for coordination, initializations, timing, and handling output.

4.4.4 Task Pools

Number of threads is fixed. It is like Master worker but workers will finish their task and then take the next task from the pool. A thread can generate new tasks and insert them into the task pool.

5 Performance of Parallel Systems

5.1 Goals and Factors

For users, to reduce the response time (time between start and end of the program). Computer managers want to have higher throughput, as long as the system is utilised well. In CS3230, we reduce wall-clock time of program by parallelizing it.

There are many factors, such as what programming model is being used, OS, compiler, architecture, memory organization, processing, execution mode.

5.2 Sequential Programs

5.2.1 Response time: Wall Clock Time

- User Time: Program. What we can fix
- System Time: OS Routines. It depends on the OS implementation
- Waiting Time: IO, Time Sharing with other programs. It depends on load of computer system

User CPU Time depends on the the translation of program statements by compiler to insttructions, which take a total number of cycles to execute. Total time is the number of total cpu cycles multipled by the cycle time of the cpu. Some instructions have different execution times, and we can use $N_{cycle}(A) = \sum_{i=1}^{n} n_i(A) \times CPI_i$ where n_i is the number of instructions of type i and CPI is the number of cpu cycles needed for that instruction.

Memory accesses should be reduced as it takes very long. Refer to CS2106. To update the formula, add in the number of cycles needed to access the memory as well.

$Time_{user}(A) = (N_{instr}(A) \times CPI(A) + N_{read,write}(A) \times R_{miss} \times N_{misscycles}) \times Time_{cycle}$

5.2.2 Measures of Throughput

MIPS THe number of instructions divided by the time * million. However, it is useless as instructions take a different amount of time to execution

MFLOPS Same but with Floating Point Operations. It is still used to rank the top 500 supercomputers but is very specific and has no differentiation between different types of FLO (addition is faster than multiplication)

5.3 Parallel Programs

Notation: $T_p(n)$, where T is the total time of parallel program on all processors, p is the number of processing units, and n is the problem size. It consists of

- Time for local computations on each processor
- Time of exchange of data
- Time for synchronization
- Waiting time due to unequal load or locked data structures.

5.3.1 Metrics

Speedup $S_p(n) = \frac{T_{seq}(n)}{T_p(n)}$. The speedup has an upper limit of p in theory, but can be even faster if cache is optimized. The best sequential time may not be known exactly, can be approximated with executing parallel on one processing unit.

Cost $C_p(n) = p \times T_p(n)$, where C is the total amount of work performed on all processors. A parallel program is cost optimal if it executes the same total number of operations as the fastest sequential program.

Efficiency $E_p(n) = \frac{T_{seq}(n)}{C_p(n)} = \frac{S_p(n)}{p}$. The ideal speedup is when the speedup is equal to p, or when efficiency is 1.

Amdahl's Law The speedup of execution is limited by the fraction of the program that cannot be parallelised. Programs cannot be fully parallel. If a program is 5% sequential, then there is no point creating more than 20 cores. Focus should instead be working on reducing the sequential threshold.

Gustafson's Law Sequential fraction is not constant with respect to problem size, and a larger problem size with the same absolute sequential means that it is more parallelizable. With large problems, f can be insignificant and thus there is a point making more cores for larger problems. Assumes that parallelizable parts parallelize perfectly, and that memory is not the main bottleneck.

5.3.2 Scalability

The scalability is the interaction between size of problem and size of parallel computer.

Scaling Constraints

- Application Oriented Scaling Properties
- Resource Oriented: Problem, Time, Memory

Arithmetic Intensity The ratio of computation to communication.

Contention occurs when many requests to a resource are made in a small window of time (if all tasks want the same resource)

5.4 Temporal Locality

Co-locate tasks that operate on the same data so that threads working on the same data structure at the saem time on the processor, and can reduce communication.

Let add() add two arrays of size n together, and mul() the same. Both of these require two loads and one store per math operation, resulting in an arithmetic intensity of 0.33. If you can fuse the data, we do not need to store intermediate results back into memory and load them agaion, allowing for a higher arithmetic intensity. Compilers nowawadays may optimize this for you.

5.5 Spatial Locality

If the cache line size is smaller than the task size, cores have to share cache lines, resulting in contention and invalidation of each other's lines. When something changes on the cache line, it is invalidated and has to be written back to memory, even if another tasks was the one that changed it.

Thus, try to make the cache line size the same as the task size. May still get some shared cache lines. This can be fixed by padding the array with dummy units.

Machines these day will also call in cache lines from neighbouring cache lines, which may be used by other cores. Compilers can guess better if your program is written in a predictable way. Accessing data in a pattern allows pre-fetcher to prefetch the correct cache line that will be used next.

5.6 Performance Analysis

5.6.1 Methodolgy

- Determine simplest parallel solution and measure performance
- Determine if performance is limited by computation, memory latency, synchronization, or others
- Establish and address this bottleneck.

5.7 Types of bottlenecks

- Instruction Rate Limited: If adding math instructions increases the time, then you should try to reduce that
- Memory bottleneck: Load the same data but remove the operations. If time is the same, memory is the issue

- Locality: Change all array accesses to A[0].
- Sync overhead: Remove all atomic operations and locks.

6 GPGPU Programming

6.1 GPU History

Problem: Since Amdahl's law is not real, we need higher processor counts. This can be solved by either

- Add more nodes with distributed computing. This results in increased communication
- Put more cores into a node, a specialized node (GPU)

CPU and GPU sit on the same motherboard, and are connected via a high speed link (PCIe or NVLink) and is much faster than talking to another node. GPU and CPU each may have their own memory.

6.1.1 Teapot

Need for GPU started with need for rendering. After modelling, need to calculate many attributes of the polyygons of the teapot. Since it is all the same computation, we can make fast processors just for performing this computation.

6.1.2 Rendering

On a 1080p screen, there are 1920 * 1080 * 3 = 6.2 million values per frame that need to be calculated at least 60 times a second, meaning that massive data parallelism is possible. After a while, GPU went from 3D game rendering to General Purpose GPUs. CUDA by NVIDIA allowed GPU to be a general purpose computing device. Compute Unified Device Architecture

In AIML classes, they will focus on using the Libraries and funtions provided by CUDA. CS3210 will focus on the programming languages used to implement them on the GPUs themselves.

6.2 Nvidia GPU Architecture

A GPU is a device, and programmes must start on the CPU first. Memory needs to be sent to the GPU memory and back (which is usually slow), so pipelining may be used to make it faster.

A streaming multiprocessor has its own memory and cache, and a connecting interface to the CPU. Each SM has multiple compute cores, such as memories (registers, cache, shared memory), and the logic needed for thread and instruction management. In GPUs, threads are managed in groups.

6.2.1 Hopper

The GPU has an L2 cache, which is connected to streaming multiprocessors each with their own L1 cache, and some streaming processors connected to it. The Hopper SM has 4 SPs, which include

- 64 INT32 CUDA Cores
- 64 FP64 CUDA Cores
- 128 FP32 CUDA Cores
- 4 Tensor Cores
- 255 Registers to allow for instant context switching. Registers are not saved but rather remain there.

CS3210 Compute Cluster has

- xgpg[0-9], xgph[0-19] - A100 (Ampere 8.6)
- xgpi[024] - H100 (Hopper 9.0)
- Will have Blackwell B200 soon.

6.3 CUDA Programming Model

Massively hardware multithreaded. It is a simple extension to C.

6.3.1 Hello World

```
__global__ void hello() {
  printf("Hello world from thread %d in block %d\n", threadIdx.x, blockIdx.x);
}

int main() {
  int blocks_per_grid = 1, threads_per_block = 1;
  hello<<blocks_per_grid, threads_per_block>>();
  cudaDeviceSynchronize();
}
```

- Compile it on the GPU with srun -G h100-47 nvcc -o hello ./hello.cu

- Execute it with srun -G h100-47 ./hello

NVCC outputs a cpu code after calling g++, and an intermediate GPU assembly code PTX. PTX is then translated to low level NVIDIA SASS code and then to binary.

The global prefix indicates the the function is the kernel of the device, and it needs to have an execution configuration (¡¡¡¿¿¿). the device prefix can also be used for other functions on the gpu, and host (default) is used for CPU. A grid can be split into blocks, and there a block can be split into threads. Threads can be indexed in 1, 2, or 3 dimensions depending on the use case.

6.3.2 CUDA Threads

Single instruction multiple threads. All threads run the kernel and the exact same program, just with different parameters. CUDA threads have little creation overhead and switch instantly. CUDA can have thousands of threads whichh scale to hundrerds of cores and thousands of threads. Threads in the block have shared memory and atomic operations, and can synchronize within the block (but not outside the block). The hardware is free to schedule thread blocks to any SM, and blocks execute in any order relative to other blocks. Can use __syncthreads() as a barrier

6.4 CUDA Execution Model

A block executes on on SM and does not migrate. Several blocks can be on one SM, limited by control and resources such as shared memory or registers

6.4.1 Warps

Each warp contains consecutive threads, and takes turns to execute on the SM until all threads finish. Scheduled by the warp scheduler. Threads in each warp start together at the same program address, and execute in lock-step. Full efficiency is realized when all threads agree on execution path. Ideally it should be the same, but if else and conditionals will cause the program to diverge and have a lower instruction throughput. As such, try to organize warps into parts where each warp will take the same path.

If the total number of threads is not a multiple of 32, then the final block will have inactive threads.

6.5 CUDA Memory Model

Each thread has 255 registers, each block has a fast shared memory in L1 cache, and all can access memory in the slow but cached GPU DRAM. The DRAM consists of

- Global Memory: Read and Write
- Constant Memory: Readonly, good for linear access

- Texture Memory: Readonly, good for 2d spatial access

CUDA has cudaMalloc, cudaMemset, and cudaFree as usualy. It also has cudaMemcpy to transfer data from the device to the host, and cudaMemcpyToSymbol to copy constant memory. managed prefix in variable declarations allows automatic computation.

6.6 Optimizing CUDA

6.6.1 Optimizing memory usage

Memory bandwidth is very expensive, and should be maximised. Different memory spaces and access patterns have different performance

- Minimizing data transfer between host and device (the slowest operation). Kernels should run on GPU even if there is no speedup over CPU, and small transfers should be batched up into one larger transfer operation. Another option is to use page-locked memory transfer (mapping memory from CPU to GPU) such that it is easier to access.

- Concurrent data transfers and execution. Use cudaMemcpyAsync instead of cudaMemcpy to overlap the transfers and computation

- Ensure global memory accesses are coalesced. The arrays of data each warp accesses should be aligned. If misaligned, a total of 32 bytes at the start and end are transferred for no reason, and need one more transaction.

- Minimize global memory access using shared memory. It has a higher bandwidth and lower latency than global memory. Are divided into banks, and addresses on different banks can be accessed simultaneously. A bank conflict occurs if two addresses of the same bank occur, and then it has to be serialized. Avoid bank conflicts. Since a warp has 32 threads and shared memory has 32 banks, this should not happen as long as the mapping from threads to banks is surjective. Except if two different threads need the same memory because there is a broadcast function.

- Minimizing bank conflicts in shared memory access. A stride of 2 results in 50% loss of load and store efficiency, as every other element is not used. And so on when increasing stride length.

6.6.2 Optimizing parallel execution

Restructuring the algorithm to expose data parallelism, and mapping to hardware to increase occupancy.

Should strike a balance between occupancy and resource utilization. The number of warps should be larger than the number of MPs, and all SPs have at least one warp. Having

more warps hides memory latency due to asynchronous execution. The number of threads per block should be a multiple of the warp size and minimally 64.

Avoid multiple programmes using the GPU as it MAY lead to multiple contexts on the GPU.

6.6.3 Optimising instruction throughput

Using high-throughput arithmetic instructions an avoiding execution paths with the same warp.

Arithmetic Instructions should trade precision for speed. Single precision floats have the best performance. Modulo and division should be replaced with bitwise operations. Use a signed loop counter, and have functions operate on int instead of char or short.

Control Flow If there are divergent warps, the threads have to wait for one to finish first. If divergence is required, try to arrange in a better way. Also try to avoid synchronisation points.

7 Cache Coherence and Memory Consistency

CPU can have a cache per core. GPu cannot do that as it has too many cores and this means that all caches need to be coherent which is too expensive.

The shared address space model is where tasks read and write to a shared varaible which is protected by locks. It is costly to scale and requires hardware support to even implement efficiently.

- Coherence is how to handle multiple cached copies when different cores read and write to the same location
- Consistency is how the order should be when multiple cores read and write to different locations.

7.1 Cache Coherence

When another core updates the value, the caches in other cores become invalid. Reading a value at an address should written the last written value by any core. Three properties required:

- Program Order
- Write Propogation
- Transaction Serialization

7.1.1 Program Order

A single core should see its memory reads and writes in program order.

7.1.2 Write Propagation

Writes from a core should become visible to other cores eventually.

7.1.3 Transaction Serialization

All writes to a location are seen in the same order bhy all processing units.

7.1.4 Maintaining Memory Coherence

Programmer must ensure programs are first correctly synchronized. Otherwise there will be data races.

Software not really used to maintain

Hardware is more common on multiprocessor systems and is known as cache coherence protocols/

Cache sizes As size increases, access time increases but cache misses decrease.

Block size Larger block size (cache lines) reduces number of blocks but replacements last longer, also increase chance of spatial locality hit.

7.1.5 Write Policy for Caches

Write through Write access is immedialtey transferred back to main memory. This is slow due to many memory accesses. Usually used less than:

Write back Writeaccess is only performed within cache and update the dirty bit to say cache is not the same anymore. This is bad because memory can contain invalid entries.

7.1.6 Tracking Cache Line Sharing Status

The best is just if each core does not used shared memory space. Cache coherence is increased memory latency. False sharing also occur if a cache line sits between two processes.

Snooping Based Each cache keeps tracks its own dirty bit. When cache is updated, set all caches to be dirty and require new data.

Directory Based A centralized location shares data. Used with NUMA.

7.2 Memory Consistency

Instructions of a program can be reordered to achieve better performance by compilers and hardware. The consistency model dictates what is allowed after reordering.

A program defines a sequence of loads and stores.

- WR: Write to X must commit before read from Y
- RR
- RW
- WW

So, sequentially consistent memory system means each processing units issues its memory operations in program order, and effect of next memory opertation must be visible to all processing units before the next one, irregardless of interleaving. This makes multiple cores operate on memory as if there is one memory.

7.2.1 Relaxed Consistency

If there are no data dependencies, then there is no need to order it as strictly. Read after Read is fine. WAR is an anti-dependency, WAR is an output dependency, and RAW is a flow dependency. This only happens when operations are on the same variable. Writing operations can occur with read operations, and should occur first so as to hide memory latency when these two memory accesses are independent.

The above is TSO. For PC, if one processor sees a change, it does not necessarily mean that all processors see this change.

RAW: Write to Read Reorderings You can read B before A is written. But other programs cannot return the value of A until the write to A is observed.

WAW: Write to Write reorderings Partial Store Ordering

8 Parallel Programming Models II

8.0.1 Distributed Memory

Distributed memory is when each processor has private memory (or separate computers). Data used by one processor is not automatically available to others. Increasing the size of the problem means that we will eventually be working on multiple computers.

More computers will provide more computational power, and also have much more memory. Also it is difficult to keep cramming more cores and memory into a single node.

Challenges

- How to spread data out
- How to pass messages
- How to write code
- How to connect
- Energy efficiency

8.1 Data (work) distribution

Parallel computing problems are commonly based on arrays of various dimensions. Having good data distribution in decomposing the arrays properly to multiple processors allows for better performance. Data allocation is now really strict because there is no more shared memory.

It depends on many factors. Trial and error works.

8.1.1 1D Arrays

Assume that there are p identical processors and n elements in array.

Blockwise Give each processor a consecutive block of n/p units. This gives a bit more locality when loading the memory.

Cyclic Give each processor every p^{th} element. May cause a bit of page faults upon loading but will likely be contiguous when loaded to device. If you do not know the size of n, this another option. There is also better load balancing in the cyclic data.

8.1.2 2D Arrays

Can use a combination of blockwise and cyclic. For example, treating one column as a unit and doing 1D. Block cyclic occurs when each processor takes up more than one row or column (block of n), and then each block is allocated cyclicly.

Checkerboard Each processor has a row and column number. An advantage is if this virtual organization matches real organization. Then the hops to another node can be figured out too. After checkerboard, the processors can be assigned blockwise, cyclic, or block-cyclic.

8.1.3 Matrix Multiplication

Less processors than elements (P=N) Each p needs to calculate one row or column of a matrix. To calculate a row, you need one row of A and all columns of B. This is better for memory access.

Same p = N^2 Unrealistic. Each p can calculate one cell of C. Rows of A can be allocated cyclicly, and columns of B as well.

8.1.4 Heat Transfer Simulation

A metal plate is modelled as a large 2d integer array, each integer representing temperature of the point of a plate. Heat transfer is iterative beased on the average of neighbouring points. Assuming p < N, and p = 4, then we can split the matrix into 4 and have each processor be in charge of the cells there. Need to update latest data across boundaries. Increasing p may decrease work per p, but increase communication and granular work for each processor.

8.2 Information Exchange

We need to ensure every processor receives and has access to the relevant information it needs to do its tasks. For shared address space, variables need to be shared. For distributed address space, we need communication operations. Race conditions and such do not happen in distributed memory.

All interactions require both parties to participate, and the programmer needs to explictly express parallelism.

Loosely Synchronous Paradigm since it is synchronous when tasks synchronize with each other, but execute asychronously otherwise.

8.3 Communication Protocols

8.3.1 Point to Point

Using send and receive. These operation can be blocking or non blocking, buffered or non buffered, and synchronous or asynchronous.

Blocking vs Non Blocking A blocking call only returns when resources used in the call may be reused safely by the programmer. A non blocking send will return immediately, and it may not be safe to change the content that is being sent. The nonblocking send returns a variable that indicates once the data has been sent. $while(!test(request))$; is a possible manual blocking before the variable being sent can be reused again.

Buffering A buffer copys the user data to send into a buffer. This buffer can accumulate until it is full, then sent. Memory copy speed is much faster than network speed, allowing the source node to update the value quicker. However, the tradeoff is space.

Synchronous A synchronous send can complete only when a matching receive has started to execute (not complete). This is non local behaviour and requires coordination with other processes. In asynchronous, it will resume computing until it gets the receive-ready request back from the receiver.

8.3.2 Global, collective communication

9 Message-passing Programming

An abstraction of a parallel computer with distributed address space. Processes exchange data by message passing using communication operations. This is done through the Message Passing Interface.

MPI is a set of library calls (directly callable in C, Fortran, via interface for Python, Java, C). The latest version is as current as June 2025. It is the go-to language for programs that are huge and need to run on many cores. (Scientific and High Performance Computing)

- Initialize the communications
- Do the communications necessary
- Exit from the message passing system

9.1 Message Passing

A programmer needs to explicit send and receive messages to use the distributed address space. For example, Fluid Simulation, Climate Modelling, and distributed machine learning. This is because a simple computer does not have enough memory for the program to run.

9.1.1 MPI Program Structure

- MPI Init()
- MPI Send() and MPI Recv()
- MPI Finalize() or MPI Abort()
- MPI Comm size: How many processors in the group
- MPI Comm rank: Get rank of process

After initializing, processes can become part of a group and send and receive to each other, until the environment is terminated. Each process has a rank within the group to identify each process. Include this library using include ¡mpi.h¿.

Hello World

```
if (rank == 0) {
  strcpy(msg, "Hello World");
  for (i = 1; i < size; i++) {
    MPI_Send(msg, 12, MPI_CHAR, i, tag, MPI_COMM_WORLD, &status);
  }
} else {
  MPI_Status status;
  MPI_Recv(msg, 12, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
}

printf("node %d : %.13\n", rank, message);
MPI_Finalize();
```

9.2 Point to Point Communication

Blocking

- MPI_Send
- MPI_Recv
- MPI_Sendrecv
- MPI_Sendrecv_replace

There is also Ssend and Rsend for synchronous.

Non blocking. It is not safe to immediately use the buffer and it needs to be checked first. Add b for buffer.

- MPI_Isend
- MPI_Irecv

Can be mixed: non blocking send can be received by blocking receive, and blocking send can be received by non blocking receive

Format A message usually has the data (start buffer address, number of elements, datatype), how to route the data (destination or source, tag (TCP), communicator)

MPI Send has buffer, count, datatype, destination, tag, and communicator

MPI Recv has buffer, count, datatype, src,, tag, communicator, and status. Src can be MPI_ANY_SOURCE or MPI_ANY_TAG if needed.

MPI$_S tatus$ is a structure with MPI Source, MPI Tag, and MPI Error

9.2.1 MPI Sending and receiving

MPI Send will call, and it may store the buffer somewhere. Only then will MPI send return. This may be the system buffer in kernel space. This system buffer is sent to the system buffer of another process, where a MPI Recv is blocking until it receives the buffer.

9.2.2 Order of Receive

Messages will be delivered in the order in which they are sent if there are only two processes. Not guaranteed if there are more than two. Assume that anything can happen on the network between the nodes.

Deadlocks can occur if process 0 waits for process 1, and vice versa. This can happen if send and receive are the first operation on both communicators. If both send first, if memory buffers are too small it will deadlock. Pair operations instead: For example, even ranks will send first, and vice versa.

9.2.3 Non Blocking Send and Receive

Can be used to avoid deadlocks, but buffers may not be safe to reuse or contain the data needed.

- MPI Test: Returns boolean if request is completed
- MPI Wait: Only return when request is completed

9.3 Process Groups and Communicators

9.3.1 Process Groups

An ordered sets of processess where each have a unique rank. A process can be a member of multiple groups with different ranks. MPI handles the representation and management of these groups. There are many MPI Group operations.

9.3.2 Communicators

A handle that processes use to communicate with each other. This provides a logical separation of processes.

Intra-communicator Supports point to point or collective operations in a single group. Is faster.

Inter-communicator Supports across groups. Use the MPI Comm operations.

9.3.3 Process Virtual Topologies

Processes can be given specific logical organizations. For example, processes usually only communicate with neighbor processes. Virtual topologies allow neighbors to be easily addressable.

9.4 Collective Communication

Operations that involve all procesess in a communicator has to be collective otherwise deadlock.

9.4.1 MPI Barrier

Synchronizes across all processes without any moving any data. All functions must call it Works just like a barrier.

9.4.2 Timings

MPI wtime and Wtick

9.4.3 MPI Bcast

THe sender sends the same data block to all other processors. The broadcast is called on all processes.

9.4.4 MPI Allgather (Multi Broadcast)

Each processor sends the same block to eveyr other processors. Data blocks are collected in rank order.

9.4.5 Scatter and Gather

The root has some data and sends based on order to other processes. Gather can then collect it back in the same order. Using a binary spanning tree method can reduce the number of total sends.

9.4.6 Single Accumulation

Each processor provides a block of data with same type and size. A reduction (binary, associative, commutative) is applied element and placed in root processor. Parallel reduction can be used again.

9.4.7 Multi-accumulation

Same as above but each processor provides for each other processor a ptoentially different data block

9.4.8 Total excahnge

Each processor performs a scatter operations.

9.4.9 Duality of Communication Operations

A communication operation can be represented by a graph of a spanning tree. Nodes are processes and directed edges are communications lines. Two communications are a duality if the same spanning tree can be used for both operations.

For example, broadcast and single accumulate

9.4.10 Stepwise Specialization

Order operatiomns to a hierarchy from most general to most specific.

10 Interconnection Networks

For distributed memory programming, how the messages travel in hardware. Even for shared memory system, there are networks between PUs and memories that are based on interconnection networks.

10.1 Interconnects

Type of interconnect determines right algorithm and performance

10.1.1 Linear Array

If all processing elements are working synchronously, sorting can be done in O(n) with odd even transposition sort. Unrealsitic to have as many PEs as numbers

10.1.2 Two Dimensional Mesh

In a normal mesh, PE at corner, edge, and center have different number of neighbours. Connecting results in a torus.

- Row Sort: Sort each row by ascending and descending on each row alternating
- Column Sort: Sort all columns by ascending order
- Do row sort again
- For N numbers, need log (N+1) phases

10.2 Topology

10.2.1 Direct Interconnections

Static, Point to Point, endpoints are of the same type.

Can be represented as graphs, with G = (V, E)

Diameter is the maximum distance between any pair of nodes. A small diameter means a small worst case distance and lower latency

Degree is the number of direct neighbour nodes. A smaller degree means few links and thus cheaper, but may be slower or have less redundancy.

Bisection Width is a measure of the network's capacity in the worst case, or the minimum number of edges that must be removed to divide the network into two equal halves.

Node connectivity is the minimum number of nodes that must fail to disconnect the network, determining the robustness of the network

Edge connectivity is the minimum number of edges that must fail to disconnect the network, determining the number of independent paths between any pairs of nodes

- Complete network has all nodes one hop away, but is very expensive and difficult to maintain
- Linear array network has large bottlenecks, and low redundancy
- Ring network: Good as long as the number of nodes is not too high
- Complete Binary Tree: Single points of failure. A fat tree adds extra links towards the root of the tree
- Two dimensional mesh: TODO
- Two dimensional torus: Same as mesh but first to last are also connected. Usually a 6D torus is used.
- Hypercubes: Hardware overload stays constant as nodes increase
- Cube Connected Cycles: Substitute each node with a cycle of k nodes, and each nodes takes one of the origin K links. Total nodes is $k2^k$

10.2.2 Indirect Interconnections

Dynamic, formed by switches. This reduces hardware costs as connections can be configured dynamically instead of just having static links. The metrics are cost (number of switches), and the number of concurrent connections.

Bus Network only one pair of devices can communicate at a time. A bus arbiter is used for coordination, and only used for small number of messages

Multistage Switching Network has serveral intermdiate switches to obtain a small distance for any arbitrary pair of devices.

Switches can have straight, crossover, upper broadcast or lower broadcast settings.

Crossbar Network has n inputs and m outputs. It is very costly as it requires nm switches.

Omega Network has one unique path for every input to output. A nn omega network has log n stages. Needs much less switches than crossbar

Butterfly Network TODO

10.3 Routing

Determines the path from source to destination. They can be based on path length, which can always choose the shortest path, and adaptive, which can be deterministic or adaptive.

10.3.1 XY Routing for 2d Meshes

Move in the X direction, then move in the Y direction. YX routing works similarly.

10.3.2 E-Cube Routing for hypercubes

THe hamming distance between source and target address is the number of hops. Start from either the MSB or LSb and hop to the neighbour. Will be completed in at most n hops if routing from the inverse address. Since the path is determinisitc, any broken node or link will prevent communication.

10.3.3 XOR Tag routing for Omega Network

Let $T = src \oplus dst$. At stage k, go straight if bit k is 0, and crossover if bit k is 1

10.4 Current Trends

- Gigabit Ethernet (Supports point ot point and broadcast, typically uses bus, star, logical bus)
- Infiniband (Supports point to point and multicast, typically uses fat tree, torus)
- Omnipath
- Custom Interconnect
- Properietary Networks

11 Energy Efficient Computing and Cloud Computing

Irish data centers used more than 21% of nations metered electricity. We are using a lot of power for computation and AI. Data centers use 2% of the worlds energy supply.

11.1 Energy and Power

Energy (J) is the capacity to do work.

- 8 Bit Add: 1 (0.03 pJ)
- 32 Bit Add: 3
- 8 Bit Mult: 6
- 32 Bit Mult: 100
- 16 Bit FP mult: 30
- 32 Bit FP Mult: 133 (4 pJ)

Power (W, Js^{-1}) is the amount of energy transferred per unittime. High power gives more performance, but results in higher cost and generates more heat.

11.1.1 Dennard Scaling

A theory that processors could always fit more transistors per unit area without using more power per unit area. Smaller transistors need more power per unit area due to current leakage. Power used by processors are mostly converted to heat. Active cooling was required in 1995 and complex cooling was required in 2010.

Need to use the power better, and reduce power usage. More logical cores were added after processor power and frequency stagnated.

Efficiency: Performance per watt, Power uSgae Effectiveness, etc.

11.2 Per Processor Efficiency

Using less power to extend battery life and reduce heat.

11.2.1 Performance per watt

Performance can be measured by GFLOPS, or by raytracing (render time, FPS), or others. Calculated by taking this score divided by CPU power.

However, performance per watt is not constant, and processors can operate at different power levels such as limiting clock frequency. Efficiency decreases as clock frequency increases since power usage increases more than the increase in performance. (Increase clock frequency is increasing voltage, which has a larger effect on power).

A processor has dynamic and static power, where static power is power used independent of work down by processor.

$P_{dynamic} = k \cdot V^2 \cdot f$, where k is a constant of the complexity of the program and hardware. The voltage and frequency are not independent.

Modern Processors adjust voltage and frequency dynamically to reduce power usage. (Such as with idle or simple background tasks.

Another method is with heterogenous cores, with smaller cores used for lower performance levels. Big Little by ARM, M4 has 10 high performance and 4 simple efficiency, Intel has P cores and E cores.

11.3 Datacenter Efficiency

Using less power to pay for less electricity and reduce emissions.

11.3.1 Motivation

Easy access from any machine and users do not have to manage hardware. Centralizing hardware is easier to manage power and cooling, and distributed jobs.

Computing at scale requires higher performance, thus more or faster computers, thus more power, thus generating more head, thus requiring more cooling, thus requiring even more power and space, and thus money.

11.3.2 GFLOPS per watt

Part of green 500. NVIDIA Grace Hopper is the most efficient superchip due to the high bandwidth CPU to GPU connection.

11.3.3 PUE

Overall energy used for compute against total energy usage. (Total Facility Energy divided by IT equipment energy. PUE is cyclic as heating is decreased during the winter. Singapore has bad PUE since it is so hot. Running larger compute loads decreases PUE, but operators may forget to measuer some other overheads. Aisles are used to separate hot and cold air. Warm water is kept at 40 degrees celsius.