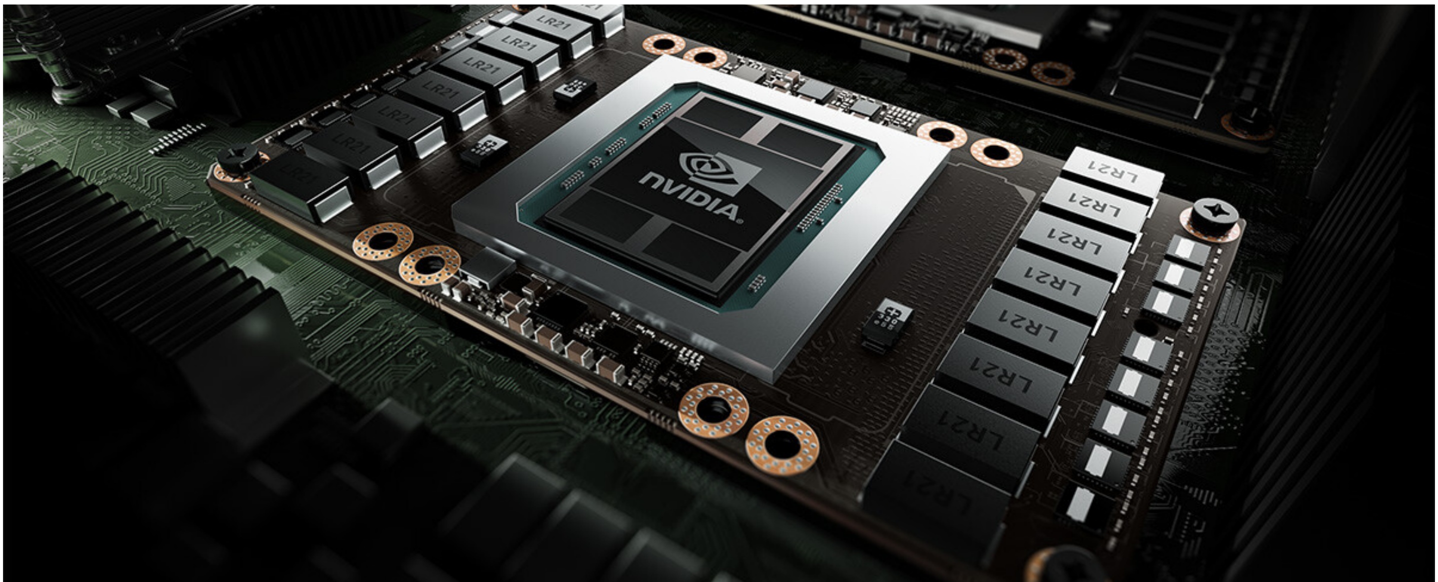
CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Computational Thinking

Haidar M. Harmanani

Spring 2020



So how do we do parallel computing?

Strategy 1: Extend Compilers

- Focus on making sequential programs parallel
- Parallelizing compiler
  - Detect parallelism in sequential program
  - Produce parallel executable program
- Advantages
  - Can leverage millions of lines of existing serial programs
  - Saves time and labor
  - Requires no retraining of programmers
  - Sequential programming easier than parallel programming
- Disadvantages
  - Parallelism may be irretrievably lost when programs written in sequential languages
  - Performance of parallelizing compilers on broad range of applications still up in air

Strategy 2: Extend Language

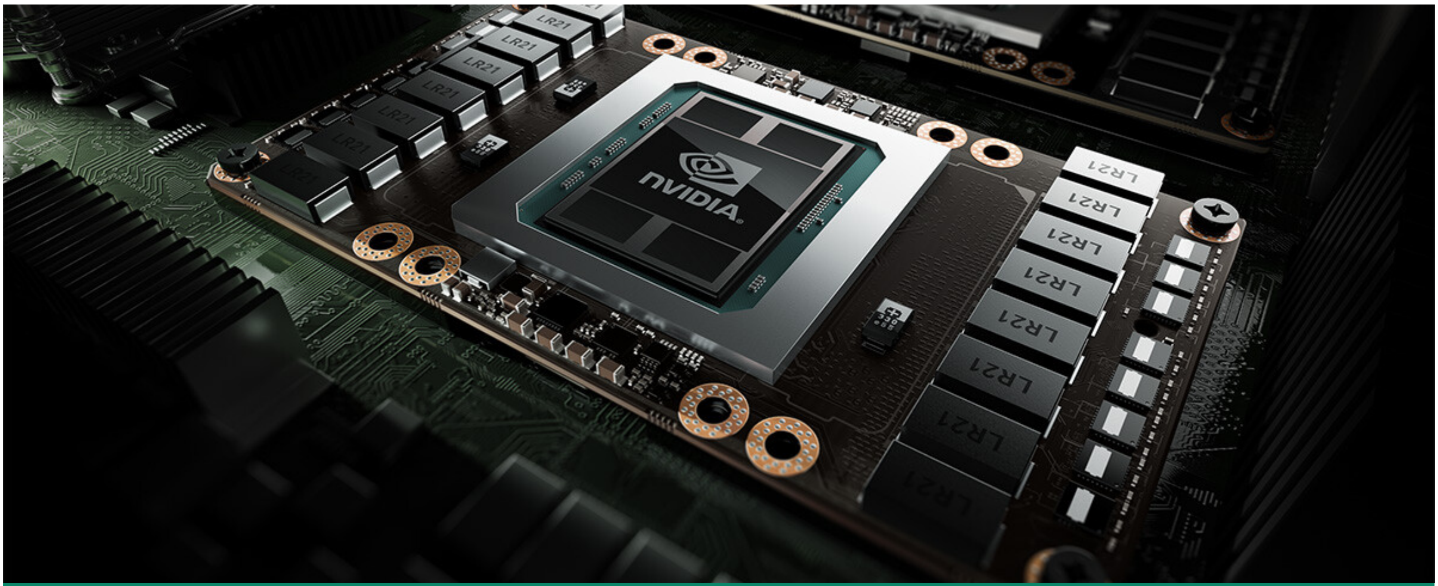
- Add functions to a sequential language
  - Create and terminate processes
  - Synchronize processes
  - Allow processes to communicate
- Advantages
  - Easiest, quickest, and least expensive
  - Allows existing compiler technology to be leveraged
  - New libraries can be ready soon after new parallel computers are available
- Disadvantages
  - Lack of compiler support to catch errors
  - Easy to write programs that are difficult to debug

Strategy 3: Add a Parallel Programming Layer

- Lower layer
  - Core of computation
  - Process manipulates its portion of data to produce its portion of result
- Upper layer
  - Creation and synchronization of processes
  - Partitioning of data among processes
- A few research prototypes have been built based on these principles

Strategy 4: Create a Parallel Language

- Develop a parallel language "from scratch"
  - occam is an example
- Add parallel constructs to an existing language
  - Fortran 90
  - High Performance Fortran
  - C*
- Advantages
  - Allows programmer to communicate parallelism to compiler
  - Improves probability that executable will achieve high performance
- Disadvantages
  - Requires development of new compilers
  - New languages may not become standards
  - Programmer resistance



Computational Thinking

Fundamentals of Parallel Computing

- Parallel computing requires that
  - The problem can be decomposed into sub-problems that can be safely solved at the same time
  - The programmer structures the code and data to solve these sub-problems concurrently
- The goals of parallel computing are
  - To solve problems in less time (strong scaling), and/or
  - To solve bigger problems (weak scaling), and/or
  - To achieve better solutions (advancing science)

The problems must be large enough to justify parallel computing and to exhibit exploitable concurrency.

Shared Memory vs. Message Passing

- We have focused on shared memory parallel programming
  - This is what CUDA (and OpenMP, OpenCL) is based on
  - Future massively parallel microprocessors are expected to support shared memory at the chip level
- The programming considerations of message passing model is quite different!
  - However, you will find parallels for almost every technique you learned in this course
  - Need to be aware of space-time constraints

Data Sharing

- Data sharing can be a double-edged sword
  - Excessive data sharing drastically reduces advantage of parallel execution
  - Localized sharing can improve memory bandwidth efficiency
- Efficient memory bandwidth usage can be achieved by synchronizing the execution of task groups and coordinating their usage of memory data
  - Efficient use of on-chip, shared storage and datapaths
- Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires more synchronization
- Many:Many, One:Many, Many:One, One:One

Synchronization

- Synchronization == Control Sharing
- Barriers make threads wait until all threads catch up
- Waiting is lost opportunity for work
- Atomic operations may reduce waiting
  - Watch out for serialization
- Important: be aware of which items of work are truly independent

Parallel Programming Coding Styles

Program Models

- **SPMD**
- Master/Worker
- Loop Parallelism
- Fork/Join

Data Models

- Shared Data
- Shared Queue
- Distributed Array

These are not necessarily mutually exclusive.

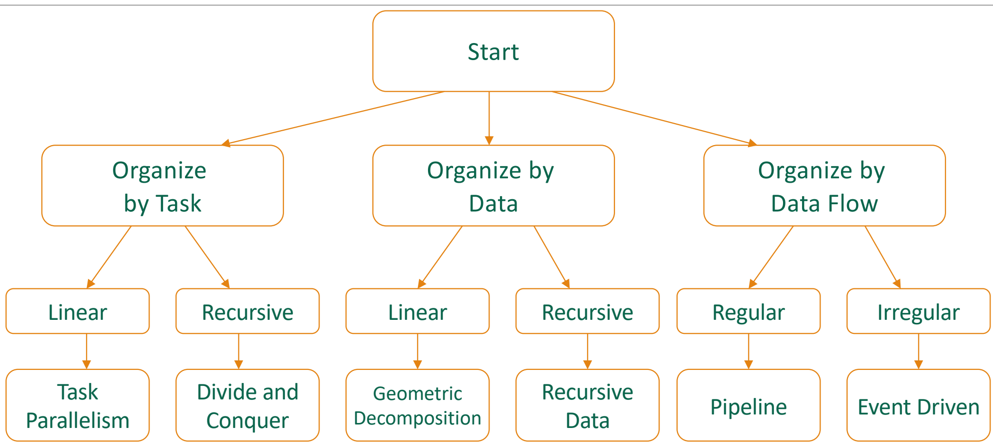
Program Models

- SPMD (Single Program, Multiple Data)
  - All PE's (Processor Elements) execute the same program in parallel, but has its own data
  - Each PE uses a unique ID to access its portion of data
  - Different PE can follow different paths through the same code
  - This is essentially the CUDA Grid model (also OpenCL, MPI)
  - SIMD is a special case – WARP used for efficiency
- Master/Worker
- Loop Parallelism
- Fork/Join

Program Models

- SPMD (Single Program, Multiple Data)
- Master/Worker (OpenMP, OpenACC, TBB)
  - A Master thread sets up a pool of worker threads and a bag of tasks
  - Workers execute concurrently, removing tasks until done
- Loop Parallelism (OpenMP, OpenACC, C++AMP)
  - Loop iterations execute in parallel
  - FORTRAN do-all (truly parallel), do-across (with dependence)
- Fork/Join (Posix p-threads)
  - Most general, generic way of creation of threads

Algorithm Structure



Mattson, Sanders, Massingill, *Patterns for Parallel Programming*

More on SPMD

- Dominant coding style of scalable parallel computing
  - MPI code is mostly developed in SPMD style
  - Many OpenMP code is also in SPMD (next to loop parallelism)
  - Particularly suitable for algorithms based on task parallelism and geometric decomposition.
- Main advantage
  - Tasks and their interactions visible in one piece of source code, no need to correlated multiple sources

SPMD is by far the most commonly used pattern for structuring massively parallel programs.

Typical SPMD Program Phases

- Initialize
  - Establish localized data structure and communication channels
- Obtain a unique identifier
  - Each thread acquires a unique identifier, typically range from 0 to N-1, where N is the number of threads.
  - Both OpenMP and CUDA have built-in support for this.
- Distribute Data
  - Decompose global data into chunks and localize them, or
  - Sharing/replicating major data structure using thread ID to associate subset of the data to threads
- Run the core computation
  - More details in next slide…
- Finalize
  - Reconcile global data structure, prepare for the next major iteration

Core Computation Phase

- Thread IDs are used to differentiate behavior of threads
  - Use thread ID in loop index calculations to split loop iterations among threads
    - Potential for memory/data divergence
  - Use thread ID or conditions based on thread ID to branch to their specific actions
    - Potential for instruction/execution divergence

Both can have very different performance results and code complexity depending on the way they are done.

Making Science Better, not just Faster

or… in other words:
There will be no Nobel Prizes or Turing Awards awarded for "just recompile" or using more threads

Conclusion: Three Options

- Good: "Accelerate" Legacy Codes
  - Recompile/Run
  - => good work for domain scientists (minimal CS required)
- Better: Rewrite / Create new codes
  - Opportunity for clever algorithmic thinking
  - => good work for computer scientists (minimal domain knowledge required)
- Best: Rethink Numerical Methods & Algorithms
  - Potential for biggest performance advantage
  - => Interdisciplinary: requires CS and domain insight
  - => Exciting time to be a computational scientist

Think, Understand… then, Program

- Think about the problem you are trying to solve
- Understand the structure of the problem
- Apply mathematical techniques to find solution
- Map the problem to an algorithmic approach
- Plan the structure of computation
  - Be aware of in/dependence, interactions, bottlenecks
- Plan the organization of data
  - Be explicitly aware of locality, and minimize global data
- Finally, write some code! (this is the easy part ☺)