GPU Basics
Types of Parallelism

S. Sundar and M. Panchatcharam

August 9, 2014
Outline

1. Task-based parallelism
2. Data-Based Parallelism
3. FLYNN’s Taxonomoy
4. Other Parallel Patterns
5. Classes of Parallel Computers
6. Simple Coding
Pipeline parallelism

- A typical OS, exploit a type of parallelism called task-based parallelism
- Example: A user read an article on a website while playing music from media player
- In terms of parallel programming, the linux operator uses pipe commands to execute this
Applications are often classified according to how often their subtasks need to synchronize or communicate with each other.

- If subtasks communicate many times per second, then it is called **fine-grained parallelism**.
- If they do not communicate many times per second, then it is called **coarser-grained parallelism**.
- If they rarely or never have to communicate, then it is called **embarrassingly parallel**.

Embarrassingly parallel applications are easiest to parallelize.
Task Parallelism

In a multiprocessor system, task parallelism is achieved when each processor executes different thread on the same or different data.

- Threads may execute same or different code
- Communication between threads takes place by passing data from one thread to next
Data Parallelism

In a multiprocessor system, each processor performs the same task on different pieces of distributed data

- Consider adding two matrices. Addition is the task on different piece of data (each element)
- It is a fine-grained parallelism
Instruction level parallelism is a measure of how many of the operations in a computer program can be performed simultaneously. The potential overlap among instructions is called instruction level parallelism.

- Hardware level works on dynamic parallelism
- Software level works on static parallelism
 Flynn’s Taxonomy is a classification of different computer architectures. Various types are as follows:

- SIMD - Single Instruction Multiple Data
- MIMD - Multiple Instruction Multiple Data
- SISD - Single Instruction Single Data
- MISD - Multiple Instruction Single Data
- Standard serial programming
- Single instruction stream on a single data
- Single-core CPU is enough
MIMD

- Today’s dual or quad-core desktop machines
- Work allocated in one of $N$ CPU cores
- Each thread has independent stream of instructions
- Hardware has the control logic for decoding separate instruction streams
SIMD

- Type of Data parallelism
- Single instruction stream at any one point of time
- Single set of logic to decode and execute the instruction stream
Many functional units perform different operations on the same data

Pipeline architectures belong to this type

Example: Horner’s Rule is an example

\[ y = \cdots (((a_n x + a_{n-1}) x + a_{n-2}) x + a_{n-3}) x + \cdots + a_1) x + a_0 \]
Loop based Pattern

- Are you familiar with loop structures?
- Types of loop?
- Entry level loop
- Exit level loop
Loop based Pattern

- Easy pattern to parallelize
- With inter-loop dependencies removed, decide splitting or partition the work between available processors
- Optimize communication between processors and the use of on chip resources
- Communication overhead is the bottleneck
- Decompose based on the number of logical hardware threads available
However, oversubscribing the number of threads leads to poor performance.

Reason: Context switching performed in software by the OS

Aware of hidden dependencies while doing existing serial implementation

Concentrate on inner loops and one or more outer loops

Best approach, parallelize only the outer loops
Loop based Pattern

- Note: Most loop can be flattened
- Reduce inner loop and outer loop to a single loop, if possible
- Example: Image processing algorithm
- X pixel axis in the inner loop and Y axis in the outer loop
- Flatten this loop by considering all pixels as a single dimensional array and iterate over image coordinates
Fork/Join Pattern

- It is a common pattern in serial programming where there are synchronization points and only certain aspects of the program are parallel.
- The serial code reaches the work that can be distributed to $P$ processors in some manner.
- It then Forks or spawns $N$ threads/processes that perform the calculation in parallel.
- These processes execute independently and finally converge or join once all the calculations are complete.
- A typical approach found in OpenMP.
Fork/Join Pattern

- Code splits into $N$ threads and later converges to a single thread again
- See figure, we see a queue of data items
- Data items are split into three processing cores
- Each data item is processed independently and later written to appropriate destination place
Fork/Join Pattern

- Typically implemented with partitioning of the data
- Serial code launches $N$ threads and divide the dataset equally between the $N$ threads
- Works well, if each packet of data takes same time to process
- If one thread takes too much time to work, it becomes single factor determining the local time
- Why did we choose three threads, Instead of six threads?
- Reality: Millions of data items attempting to fork million threads will cause almost all OS to fail
- OS applies "fair scheduling policy"
Fork/Join Pattern

- Programmer and many multithreaded libraries will use the number of logical processor threads available as number of processes to fork.
- CPU threads are so expensive to create, destroy and utilize.
- Fork/join pattern is useful when there is an unknown amount of concurrency available in a problem.
- Traversing a tree structure may fork additional threads when it encounters another node.
Divide and Conquer Pattern

- A pattern for breaking down (divide) large problems into smaller sections each of which can be conquered
- Useful with recursion
- Example: Quick Sort
- Quick sort recursively partitions the data into two sets, above pivot point and below pivot point
Divide and Conquer Pattern

3 7 8 5 2 1 9 5 4

3 7 8 4 2 1 9 5 5

3 4 2 7 8 1 9 5 5

3 4 2 1 5 7 9 8 5

3 4 2 1 5 5 9 8 7
Implicit Parallelism

It is characteristic of a programming language that allows a compiler or interpreter to automatically exploit the parallelism inherent to the computations expressed by some of the language’s constructs.

In other words, it is a compiler level parallelism, such as openmp flags or -O3 flags.
Implicit Parallelism

Advantages:
- Need not worry about task division or communication
- Focus on the problem rather than parallelization
- Substantial improvement of programmer’s productivity
- Add simplicity and clarity to the program
Implicit Parallelism

Disadvantages:

- Programmer loose the control over the parallel execution of the program
- Gives less than optimal parallel efficiency
- Debugging is difficult
- Every program has some parallel and serial logic
Implicit Parallelism

It is the representation of concurrent optimizations by means of primitive in the form of special-purpose directives or function calls.

- Related to process synchronization, communication, partitioning
- Absolute programmer control over the parallel execution
- Skilled programmer takes advantage of explicit parallelism to produce very efficient code
- However, it is difficult to do explicit parallelism
- Reason: Extra work in planning decomposition/partition, synchronization of concurrent process
Classes of Parallel Computers

- Multicore computing
- Symmetric multiprocessing
- Distributed computing
- Cluster Computing
- Massive parallel computing
- Grid computing
Classes of Parallel Computers

- Reconfigurable computing with field programmable gate arrays (FPGA)
- General Purpose computing on Graphics Processing Units (GPGPU)
- Application Specific integrated circuits
- Vector Processors
Multicore computing

- Contains multiple execution units on the same chip
- Multiple instructions per cycle from multiple instruction streams
- Example: Intel Quad-Core

Advantages:
- Multiple CPU cores on the same die allows the cache coherency to operate at much higher clock rate
- It allows higher performance at lower energy

Disadvantages:
- Difficult to manage thermal power
- Two processing cores shares the same bus and memory limiting the real-world performance
Symmetric Multiprocessing (SMP)

- SMP is a system with multiple identical processors that share memory and connect via bus
- SMPs don’t compromise more than 32 processors
- SMPs are cost effective if sufficient amount of memory bandwidth exists
- Example: Intel’s Xeon machine
Distributed Computing

It is a distributed memory computer system in which the processing elements are connected by a network

Examples:
- Telephone networks
- World wide web/Internet
- Aircraft control systems
- Multiplayer online games
- Grid Computing
Cluster Computing

Cluster

Cluster is a group of loosely coupled computers that work together closely, considered as a single computer

- Consists of multiple standalone computers connected by a network
- Load balancing is difficult if the cluster machines are not symmetric
- Examples: Beowulf cluster, TCP/IP LAN
- Top500 supercomputers are clusters
Massively Parallel Computing

**MPP**

It is a single computer with many networked processors

- Similar to clusters, but specialized with interconnect networks
- Larger than clusters
- Has more than 100 processors
- Each CPU has its own memory and OS/applications
- Communicates via high-speed interconnect
- Example: Blue Gene/L, the fifth fastest super computer
NUMA

It stands for Non-Uniform Memory Access and is a special type of shared memory architecture where access times of different memory locations by a processor may vary as may also access times to the same memory location by different processors.

- It is known as a tightly coupled form of cluster computing.
- Influences the memory access performance.
- Most of the current OS such as Windows 7, 8 support NUMA.
- Linux kernel 2.5 supports NUMA.
Modern CPUs work faster than the main memory they use.

All of the processors have equal access to the memory I/O in the system (see figure).

Special attention require to write a software run on NUMA.
Parallel Random Access Machine

- It is a shared memory abstract machine
- Read/write conflicts in accessing the same shared memory location simultaneously are resolved by the following categories
  - EREW-Exclusive Read, exclusive write. Every memory cell can read/write by only one processor at a time
  - CREW-Concurrent Read, exclusive write. Multiple processors can read a memory but only one can write at a time
  - CRCW-Concurrent Read, concurrent write. Multiple processors can read and write
FAQ on Parallel Computing
FAQ

- **Shared Memory architecture**: Single address space is visible to all execution threads
- **Task-latency**: The time taken for a task to complete since a request for it is made
- **Task-throughput**: The number of tasks completed in a given time
- **Speed up**: The ratio of some performance metric obtained using a single processor with that obtained using a set of parallel processors
- **Parallel Efficiency**: The speed-up per processor
FAQ

- Maximum time speed up possible according to Amdhal’s law: 
  \[
  \frac{1}{f_s}, \quad \text{where } f_s \text{ is inherently sequential fraction of the time taken}
  \]
  by the best sequential of the task (Prove!)

- **Cache Coherence:** Different processors maintain their own local caches. That is multiple copies of the same data available. Coherence implies that access to the local copies behave similarly to access from the local copy–apart from the time to access.

- **False Sharing:** Sharing of a cache line by distinct variables. If such variables are not accessed together, the un-accessed variable is unnecessarily brought into cache along with the accessed variable.
Finally, let us look at an unsolved problem (open Problem) in Parallel Computing

**NC=P**

The class NC is the set of decision problems decidable in polylogarithmic time on parallel computer with a polynomial number of processors. That is, a problem is in NC, if there exists constants $c$ and $k$ such that it can be solved in time $O(\log^c n)$ using $O(n^k)$ parallel processors.
NC=\text{P} Problem

\textbf{NC=\text{P}}

\begin{itemize}
  \item \textit{P} : tractable or efficiently solvable problem or decision problems that can be solved by a deterministic turing machine using polynomial time
  \item \textit{NC} : problems can be efficiently solved on a parallel computer
  \item \textit{NC} \subset \textit{P} as polylogarithmic parallel computations can be simulated by polynomial time sequential ones
\end{itemize}

\textbf{Open Problem:} Does \textit{NC} = \textit{P}
NC=P Problem

NC problem is known to include many problems including:

- Integer addition, multiplication and division
- Matrix multiplication, determinant, inverse, rank
- Polynomial GCD, Sylvester Matrix
Simple Coding
Simple Vector Addition

Vector addition

```c
for (i=0; i<N; i++)
c[i] = a[i] + b[i];
```

**Explanation**

- **Step 1:** \( c[0] = a[0] + b[0] \)
- **Step 2:** \( c[1] = a[1] + b[1] \)
  
  ...  
  
- **Step N:** \( c[N-1] = a[N-1] + b[N-1] \)

- Each addition is independent of others
Parallelization in CPU

- Assume you have 4 processors and \( N \) is divisible by 4

**Explanation**

- **Step 1:**
  \[
  c[0:3] = a[0:3] + b[0:3]
  \]

- **Step 2:**
  \[
  \]

- **Step \( N/4 \):**
  \[
  \]

- 4 elements are added simultaneously
Parallelization in Intel OpenMP

OpenMP code

```c
#include <omp.h>
#pragma omp for schedule (static, chunk)
for (i=0; i<N; i++)
c[i]=a[i]+b[i];
```

- Intel OpenMP cares the parallelization
- Autoparallelization in Intel
THANK YOU