CS 677: Parallel Programming for Many-core Processors
Lecture 9

Instructor: Philippos Mordohai
Webpage: www.cs.stevens.edu/~mordohai
E-mail: Philippos.Mordohai@stevens.edu
Outline

• Computational Thinking
  – Chapter 17 in K&H, 3rd edition
Objective

- To provide you with a framework based on the techniques and best practices used by experienced parallel programmers for
  - Thinking about the problem of parallel programming
  - Addressing performance and functionality issues in your parallel program
  - Using or building useful tools and environments
  - Understanding case studies and projects
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, and/or
  – To solve bigger problems, and/or
  – To achieve better solutions

The problems must be large enough to justify parallel computing and to exhibit exploitable concurrency.
Key Parallel Programming Steps

1) To find the concurrency in the problem
2) To structure the algorithm so that concurrency can be exploited
3) To implement the algorithm in a suitable programming environment
4) To execute and tune the performance of the code on a parallel system

Unfortunately, these have not been separated into levels of abstractions that can be dealt with independently.
Amdahl’s Law

• “The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.”

• Example
  – 95% of original execution time can be sped up by 100x on GPU
  – Speed up for entire application:

\[
\frac{1}{(5\% + \frac{95\%}{100})} = \frac{1}{5\% + 0.95\%} = \frac{1}{5.95\%} = 17x
\]
Challenges of Parallel Programming

• Finding and exploiting concurrency often requires looking at the problem from a non-obvious angle
  – Computational thinking

• Dependences need to be identified and managed
  – The order of task execution may change the answers
    • Obvious: One step feeds result to the next steps
    • Subtle: numeric accuracy may be affected by ordering steps that are logically parallel with each other

• Performance can be drastically reduced by many factors
  – Overhead of parallel processing
  – Load imbalance among processor elements
  – Inefficient data sharing patterns
  – Saturation of critical resources such as memory bandwidth
Shared Memory vs. Message Passing

• We have focused on shared memory parallel programming
  – This is what CUDA is based on
  – Future massively parallel microprocessors are expected to support shared memory at the chip level
  – This is different than global address space (single pointer space)

• The programming considerations of message passing model is quite different!
  – See MPI (Message Passing Interface)
Finding Concurrency in Problems

• Identify a decomposition of the problem into sub-problems that can be solved simultaneously
  – A task decomposition that identifies tasks for potential concurrent execution
  – A data decomposition that identifies data local to each task
  – A way of grouping tasks and ordering the groups to satisfy temporal constraints
  – An analysis on the data sharing patterns among the concurrent tasks
  – A design evaluation that assesses the quality of the choices made in all the steps
Finding Concurrency - The Process

This is typically an iterative process. Opportunities exist for dependence analysis to play earlier role in decomposition.
Task Decomposition

• Many large problems can be naturally decomposed into tasks - CUDA kernels are largely tasks
  – The number of tasks used should be adjustable to the execution resources available
  – Each task must include sufficient work in order to compensate for the overhead of managing their parallel execution
  – Tasks should maximize reuse of sequential program code to minimize effort

“In an ideal world, the compiler would find tasks for the programmer. Unfortunately, this almost never happens.”
  - Mattson, Sanders, Massingill

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE408, University of Illinois, Urbana-Champaign
Task Decomposition Example - Square Matrix Multiplication

- $P = M \times N$ of $\text{WIDTH} \times \text{WIDTH}$
  - One natural task (sub-problem) produces one element of $P$
  - All tasks can execute in parallel in this example.
Task Decomposition Example - Molecular Dynamics

• Simulation of motions of a large molecular system
• For each atom, there are natural tasks to calculate
  – Vibrational forces
  – Rotational forces
  – Neighbors that must be considered in non-bonded forces
  – Non-bonded forces
  – Update position and velocity
  – Misc physical properties based on motions
• Some of these can go in parallel for an atom

Often there are multiple ways to decompose any given problem.
Task Ordering Example: Molecular Dynamics

Vibrational and Rotational Forces → Neighbor List
   ↓
Non-bonded Force → Update atomic positions and velocities
   ↓
Next Time Step

Complex computation involving many atoms
Data Decomposition

• The most compute intensive parts of many large problem manipulate a large data structure
  • Similar operations are being applied to different parts of the data structure, in a mostly independent manner
  • This is what CUDA is optimized for

• The data decomposition should lead to
  • Efficient data usage by tasks within the partition
  • Few dependencies across the tasks that work on different partitions
  • Adjustable partitions that can be varied according to the hardware characteristics
Task Grouping

• Sometimes natural tasks of a problem can be grouped together to improve efficiency
  – Reduced synchronization overhead - all tasks in the group can use a barrier to wait for a common dependence
  – All tasks in the group efficiently share data loaded into a common on-chip, shared storage (Shared Memory)
  – Grouping and merging dependent tasks into one task reduces need for synchronization
  – CUDA thread blocks are task grouping examples
Task Grouping Example - Square Matrix Multiplication

- Tasks calculating a P sub-block
  - Extensive input data sharing, reduced memory bandwidth using Shared Memory
  - All synched in execution
Task Ordering

- Identify the data and resources required by a group of tasks before they can be executed
  - Find the task group that creates them
  - Determine a temporal order that satisfies all data constraints
Data Sharing

• Data sharing can be a double-edged sword
  – Excessive data sharing can drastically reduce 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

• Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires synchronization
Data Sharing Example - Matrix Multiplication

• Each task group will finish usage of each sub-block of N and M before moving on
  – N and M sub-blocks loaded into Shared Memory for use by all threads of a P sub-block
  – Amount of on-chip Shared Memory strictly limits the number of threads working on a P sub-block

• Read-only shared data can be more efficiently accessed as Constant or Texture data
Data Sharing Example - Molecular Dynamics

- The atomic coordinates
  - Read-only access by the neighbor list, bonded force, and non-bonded force task groups
  - Read-write access for the position update task group

- The force array
  - Read-only access by position update group
  - Accumulate access by bonded and non-bonded task groups

- The neighbor list
  - Read-only access by non-bonded force task groups
  - Generated by the neighbor list task group
Key Parallel Programming Steps

1) To find the concurrency in the problem
2) To structure the algorithm to translate concurrency into performance
3) To implement the algorithm in a suitable programming environment
4) To execute and tune the performance of the code on a parallel system
Algorithm

• A step by step procedure that is guaranteed to terminate, such that each step is precisely stated and can be carried out by a computer
  – Definiteness - the notion that each step is precisely stated
  – Effective computability - each step can be carried out by a computer
  – Finiteness - the procedure terminates

• Multiple algorithms can be used to solve the same problem
  – Some require fewer steps
  – Some exhibit more parallelism
  – Some have larger memory footprint than others
Choosing Algorithm Structure

Start

Organize by Task
- Linear
  - Task Parallelism
- Recursive
  - Divide and Conquer

Organize by Data
- Linear
  - Geometric Decomposition
- Recursive
  - Recursive Data

Organize by Data Flow
- Regular
  - Pipeline
- Irregular
  - Event Driven
Tiled (Stenciled) Algorithms are Important for Geometric Decomposition

• A framework for memory data sharing and reuse by increasing data access locality.
  – Tiled access patterns allow small cache/scartchpad memories to hold on to data for re-use.
  – For matrix multiplication, a 16X16 thread block perform $2 \times 256 = 512$ float loads from device memory for $256 \times (2 \times 16) = 8,192$ mul/add operations.

• A convenient framework for organizing threads (tasks)
Increased Work per Thread for even more locality

- Each thread computes two elements of $P_{d_{sub}}$
- Reduced loads from global memory ($Md$) to shared memory
- Reduced instruction overhead
  - More work done in each iteration
Double Buffering
- a frequently used algorithm pattern

• One could double buffer the computation, getting better instruction mix within each thread
  – This is classic software pipelining in ILP compilers

<table>
<thead>
<tr>
<th>Loop {</th>
<th>Load next tile from global memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Loop {</td>
</tr>
<tr>
<td></td>
<td>Load current tile to shared memory</td>
</tr>
<tr>
<td></td>
<td>syncthreads()</td>
</tr>
<tr>
<td>Load current tile to shared memory</td>
<td>Compute current tile</td>
</tr>
<tr>
<td>syncthreads()</td>
<td>syncthreads()</td>
</tr>
<tr>
<td>}</td>
<td>Load next tile from global memory</td>
</tr>
<tr>
<td></td>
<td>Compute current tile</td>
</tr>
<tr>
<td></td>
<td>syncthreads()</td>
</tr>
</tbody>
</table>
Double Buffering

- Deposit blue tile from register into shared memory
- Syncthreads
- Load orange tile into register
- Compute Blue tile
- Deposit orange tile into shared memory
- ....
(a) Direct summation
At each grid point, sum the electrostatic potential from all charges

(b) Cutoff summation
Electrostatic potential from nearby charges summed; spatially sort charges first

(c) Cutoff summation using direct summation kernel
Spatially sort charges into bins; adapt direct summation to process a bin
Cut-Off Summation Restores Data Scalability

Scalability and Performance of different algorithms for calculating electrostatic potential map.
Parallel Programming Coding Styles - Program and Data Models

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 PEs (Processor Elements) execute the same program in parallel, but each has its own data
  - Each PE uses a unique ID to access its portion of data
  - Different PEs can follow different paths through the same code
  - This is essentially the CUDA Grid model (also MPI)
  - SIMD is a special case - WARP

- **Master/Worker (CUDA Streams)**
- **Loop Parallelism (OpenMP)**
- **Fork/Join (Posix p-threads)**
Program Models

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

• API that supports shared memory multiprocessing in C, C++ and Fortran

• A master thread forks a specified number of slave threads and the system divides a task among them

```c
int main(int argc, char **argv) {
    int a[100000];

    #pragma omp parallel for
    int i;
    for (i = 0; i < N; i++)
        a[i] = 2 * i;

    return 0;
}
```
Message Passing Interface (MPI)

- Standardized, portable and language-independent message passing system
- Supports point-to-point and collective communication

```c
int array[100];
int root, total_p, *receive_array;

MPI_Comm_size(comm, &total_p);
receive_array = malloc(total_p*100*sizeof(*receive_array));
MPI_Gather(array, 100, MPI_INT, receive_array, 100, MPI_INT, root, comm);
```
Algorithm Structures vs. Program Models

<table>
<thead>
<tr>
<th></th>
<th>Task Parallel.</th>
<th>Divide/Conquer</th>
<th>Geometric Decomp.</th>
<th>Recursive Data</th>
<th>Pipeline</th>
<th>Event-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPMD</td>
<td>☺☺☺☺</td>
<td>☺☺☺</td>
<td>☺☺☺</td>
<td>☺</td>
<td>☺☺☺☺</td>
<td>☺</td>
</tr>
<tr>
<td>Loop Parallel</td>
<td>☺☺☺</td>
<td>☺</td>
<td>☺☺☺</td>
<td></td>
<td></td>
<td>☺</td>
</tr>
<tr>
<td>Master/Worker</td>
<td>☺☺☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Fork/Join</td>
<td>☺</td>
<td>☺☺☺</td>
<td>☺☺</td>
<td>☺</td>
<td>☺☺☺☺</td>
<td>☺</td>
</tr>
</tbody>
</table>

Source: Mattson, et al
# Program Models vs. Architectures

<table>
<thead>
<tr>
<th></th>
<th>OpenMP</th>
<th>MPI</th>
<th>CUDA/OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPMD</td>
<td>☺☺☺☺</td>
<td>☺☺☺</td>
<td>☺☺☺☺☺</td>
</tr>
<tr>
<td>Loop Parallel</td>
<td>☺☺☺☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Master/Slave</td>
<td>☺☺</td>
<td>☺☺☺</td>
<td>☺☺</td>
</tr>
<tr>
<td>Fork/Join</td>
<td>☺☺☺☺</td>
<td></td>
<td>☺</td>
</tr>
</tbody>
</table>
More on SPMD

• Dominant coding style of scalable parallel computing
  – MPI code is mostly developed in SPMD style
  – A lot of 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 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 \( N \), 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
  - Share/replicate 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
  – Use thread ID or conditions based on thread ID to branch to specific actions

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

• Assume
  – The computation being parallelized has 1,000,000 iterations.

• Sequential code:

  num_steps = 1000000;

  for (i=0; i< num_steps, i++ ) {
    ...
  }
SPMD Code Version 1

• Assign a chunk of iterations to each thread
  – The last thread also finishes up the remaining iterations

    //num_steps = 1000000;
    ...
    i_start = my_id * (num_steps/num_threads);
    i_end = i_start + (num_steps/num_threads);
    if (my_id == (num_threads-1)) i_end = num_steps;

    for (i = i_start; i < i_end; i++) {
        ....
    }

    //Reconciliation of results across threads if necessary
Problems with Version 1

- The last thread executes more iterations than others
- The number of extra iterations is up to the total number of threads - 1
  - This is not a big problem when the number of threads is small
  - When there are thousands of threads, this can create serious load imbalance problems
- Also, the extra if statement is a typical source of “branch divergence” in CUDA programs
SPMD Code Version 2

• Assign one more iteration to some of the threads

```c
int rem = num_steps % num_threads;
i_start = my_id * (num_steps/num_threads);
i_end = i_start + (num_steps/num_threads);

if (rem != 0) {
    if (my_id < rem) {
        i_start += my_id;
i_end += (my_id +1);
    }
    else {
        i_start += rem;
i_end += rem;
    }
```

Less load imbalance
More branch divergence
SPMD Code Version 3

• Use cyclic distribution of iteration

    num_steps = 1000000;

    for (i = my_id; i < num_steps; i+= num_threads) {
        ....
    }

    Less load imbalance

    Loop branch divergence in the last Warp

    Data padding further eliminates divergence
Comparing the Three Versions

Version 1

ID=0    ID=1    ID=2    ID=3

Version 2

ID=0    ID=1    ID=2    ID=3

Version 3

ID=0    ID=1    ID=2    ID=3

Padded version 3 may be best for some data access patterns