DATA/THREAD LEVEL PARALLELISM

Mahdi Nazm Bojnordi
Assistant Professor
School of Computing
University of Utah
Overview

- Announcement
  - Tonight: Homework 5 is due
  - Reminder: we will drop one of your HW with the least grade

- This lecture
  - Data level parallelism
    - Graphics processing unit
  - Thread level parallelism
Flynn’s Taxonomy

- Data vs. instruction streams

Instruction Stream

Data Stream
Flynn’s Taxonomy

- Data vs. instruction streams

Single-Instruction, Single Data (SISD) uniprocessors
Flynn’s Taxonomy

- Data vs. instruction streams

- Single-Instruction, Single Data (SISD) uniprocessors
- Single-Instruction, Multiple Data (SIMD) vector processors
Flynn’s Taxonomy

- Data vs. instruction streams

**Instruction Stream**

- **Single**
  - Single-Instruction, Single Data (SISD) uniprocessors

- **Multiple**
  - Multiple-Instruction, Single Data (MISD) systolic arrays
  - Multiple-Instruction, Multiple Data (SIMD) vector processors
  - Multiple-Instruction, Multiple Data (MIMD) multicore
Graphics Processing Unit

- Initially developed as graphics accelerators
  - one of the densest compute engines available now
- Many efforts to run non-graphics workloads on GPUs
  - general-purpose GPUs (GPGPUs)
- C/C++ based programming platforms
  - CUDA from NVidia and OpenCL from an industry consortium
- A heterogeneous system
  - a regular host CPU
  - a GPU that handles CUDA (may be on the same CPU chip)
Graphics Processing Unit

- Simple in-order pipelines that rely on thread-level parallelism to hide long latencies
- Many registers (~1K) per in-order pipeline (lane) to support many active warps
Why GPU Computing?

Source: NVIDIA

---

**Figure:**
- **GFlops/sec** graph showing performance improvements from 2003 to 2010 for different CPU models and NVIDIA GPU models.
- **GBytes/sec** graph showing performance improvements from 2003 to 2010 for different CPU models and NVIDIA GPU models.

Legend:
- Single Precision: NVIDIA GPU
- Double Precision: NVIDIA GPU
- Single Precision: x86 CPU
- Double Precision: x86 CPU
- NVIDIA GPU ECC off
- X86 CPU

**Source:** NVIDIA
The GPU Architecture

- SIMT – single instruction, multiple threads
  - GPU has many SIMT cores
- Application ➔ many thread blocks (1 per SIMT core)
- Thread block ➔ many warps (1 warp per SIMT core)
- Warp ➔ many in-order pipelines (SIMD lanes)
GPU Computing

- GPU as an accelerator in scientific applications
GPU Computing

- Low latency or high throughput?

**CPU**
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

**GPU**
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation
GPU Computing

- Low latency or high throughput
CUDA Programming Model

- **Step 1**: substitute library calls with equivalent CUDA library calls
  - `saxpy ( ... ) \rightarrow \text{cublasSaxpy} ( ... )`
  - Single precision alpha x plus y \((z = \alpha x + y)\)

- **Step 2**: manage data locality
  - `cudaMalloc()`, `cudaMemcpy()`, etc.

- **Step 3**: transfer data between CPU and GPU
  - Get and set functions

- Rebuild and link the CUDA-accelerated library
  - `nvcc myobj.o -l cublas`
Example: SAXPY Code

```c
int N = 1 << 20;

// Perform SAXPY on 1M elements: y[] = a*x[] + y[]
saxpy(N, 2.0, x, 1, y, 1);
```
Example: CUDA Lib Calls

```c
int N = 1 << 20;

// Perform SAXPY on 1M elements: d_y[] = a*d_x[] + d_y[]
古人cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
```
Example: Initialize CUDA Lib

```c
int N = 1 << 20;

cublasInit();

// Perform SAXPY on 1M elements: d_y[] = a * d_x[] + d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasShutdown();
```
int N = 1 << 20;

cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void*)&d_y);

// Perform SAXPY on 1M elements: d_y[] = a*d_x[] + d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
Example: Transfer Data

```c
int N = 1 << 20;

cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void*)&d_y);

cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements: d_y[] = a*d_x[] + d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);

cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
```
Compiling CUDA

- Call `nvcc`
- Parallel Threads eXecution (PTX)
  - Virtual machine and ISA
- Two stage
  - 1. PTX
  - 2. device-specific binary object
Memory Hierarchy

- Throughput-oriented main memory
  - Graphics DDR (GDDR)
    - Wide channels: 256 bit
    - Lower clock rate than DDR
  - 1.5MB shared L2
  - 48KB read-only data cache
    - Compiler controlled
  - Wide buses
Thread Level Parallelism
Flynn’s Taxonomy

- **Forms of computer architectures**

  - **Instruction Stream**
    - **Single**
      - Single-Instruction, Single Data (SISD) uniprocessors
    - **Multiple**
      - Multiple-Instruction, Single Data (MISD) systolic arrays
  - **Data Stream**
    - **Single**
      - Single-Instruction, Multiple Data (SIMD) vector processors
    - **Multiple**
      - Multiple-Instruction, Multiple Data (MIMD) multiprocessors
Basics of Threads

- **Thread** is a single sequential flow of control within a program including instructions and state
  - Register state is called *thread context*

- A program may be single- or multi-threaded
  - Single-threaded program can handle one task at any time

- **Multitasking** is performed by modern operating systems to load the context of a new thread while the old thread’s context is written back to memory
Thread Level Parallelism (TLP)

- Users prefer to execute multiple applications
  - Piping applications in Linux
    - `gunzip -c foo.gz | grep bar | perl some-script.pl`
  - Your favorite applications while working in office
    - Music player, web browser, terminal, etc.
- Many applications are amenable to parallelism
  - Explicitly multi-threaded programs
    - Pthreaded applications
  - Parallel languages and libraries
    - Java, C#, OpenMP
Thread Level Parallel Architectures

- Architectures for exploiting thread-level parallelism

**Hardware Multithreading**

- Multiple threads run on the same processor pipeline
- Multithreading levels
  - Coarse grained multithreading (CGMT)
  - Fine grained multithreading (FGMT)
  - Simultaneous multithreading (SMT)

**Multiprocessing**

- Different threads run on different processors
- Two general types
  - Symmetric multiprocessors (SMP)
    - Single CPU per chip
  - Chip Multiprocessors (CMP)
    - Multiple CPUs per chip
Hardware Multithreading
Hardware Multithreading

- **Observation**: CPU become idle due to latency of memory operations, dependent instructions, and branch resolution
- **Key idea**: utilize idle resources to improve performance
  - Support multiple thread contexts in a single processor
  - Exploit thread level parallelism
- **Challenge**: the energy and performance costs of context switching
Coarse Grained Multithreading

- Single thread runs until a costly stall—e.g. last level cache miss
- Another thread starts during stall for first thread
  - Pipeline fill time requires several cycles!
- At any time, only one thread is in the pipeline
- Does not cover short stalls
- Needs hardware support
  - PC and register file for each thread
Coarse Grained Multithreading

Superscalar vs. CGMT

Conventional Superscalar

Coarse Grained Multithreading
Two or more threads interleave instructions
- Round-robin fashion
- Skip stalled threads

Needs hardware support
- Separate PC and register file for each thread
- Hardware to control alternating pattern

Naturally hides delays
- Data hazards, Cache misses
- Pipeline runs with rare stalls

Does not make full use of multi-issue architecture
Fine Grained Multithreading

- CGMT vs. FGMT

Coarse Grained Multithreading

- FU1
- FU2
- FU3
- FU4

Fine Grained Multithreading

- FU1
- FU2
- FU3
- FU4
Simultaneous Multithreading

- Instructions from multiple threads issued on same cycle
  - Uses register renaming and dynamic scheduling facility of multi-issue architecture
- Needs more hardware support
  - Register files, PC’s for each thread
  - Temporary result registers before commit
  - Support to sort out which threads get results from which instructions
- Maximizes utilization of execution units
Simultaneous Multithreading

- FGMT vs. SMT

Fine Grained Multithreading

Simultaneous Multithreading
Multiprocessing
Symmetric Multiprocessors

- Multiple CPU chips share the same memory
- From the OS’s point of view
  - All of the CPUs have equal compute capabilities
  - The main memory is equally accessible by the CPU chips
- OS runs every thread on a CPU
- Every CPU has its own power distribution and cooling system

AMD Opteron
Chip Multiprocessors

- Can be viewed as a simple SMP on single chip
- CPUs are now called cores
  - One thread per core
- Shared higher level caches
  - Typically the last level
  - Lower latency
  - Improved bandwidth
- Not necessarily homogenous cores!

Intel Nehalem (Core i7)
Why Chip Multiprocessing?

- CMP exploits parallelism at lower costs than SMP
  - A single interface to the main memory
  - Only one CPU socket is required on the motherboard
- CMP requires less off-chip communication
  - Lower power and energy consumption
  - Better performance due to improved AMAT
- CMP better employs the additional transistors that are made available based on the Moore’s law
  - More cores rather than more complicated pipelines
**Efficiency of Chip Multiprocessing**

- **Ideally**, $n$ cores provide $nx$ performance
- **Example**: design an ideal dual-processor
  - **Goal**: provide the same performance as uniprocessor

<table>
<thead>
<tr>
<th></th>
<th>Uniprocessor</th>
<th>Dual-processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>Voltage</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>Execution Time</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dynamic Power</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>Dynamic Energy</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>Energy Efficiency</td>
<td>1</td>
<td>?</td>
</tr>
</tbody>
</table>
Efficiency of Chip Multiprocessing

- Ideally, \( n \) cores provide \( nx \) performance
- Example: design an ideal dual-processor

**Goal**: provide the same performance as uniprocessor

\[ f \propto V \quad \text{and} \quad P \propto V^3 \Rightarrow V_{\text{dual}} = 0.5V_{\text{uni}} \Rightarrow P_{\text{dual}} = 2 \times 0.125P_{\text{uni}} \]

<table>
<thead>
<tr>
<th></th>
<th>Uniprocessor</th>
<th>Dual-processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>Voltage</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>Execution Time</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dynamic Power</td>
<td>1</td>
<td>2x0.125</td>
</tr>
<tr>
<td>Dynamic Energy</td>
<td>1</td>
<td>2x0.125</td>
</tr>
<tr>
<td>Energy Efficiency</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
Challenges
A sequential application runs as a single thread

Kernel Function:

```c
void kern (int start, int end) {
    int i;
    for(i=start; i<=end; ++i) {
    }
}
```

Memory:

A

1   ...   n

Single Thread

```c
main() {
    ...
    kern (1, n);
    ...
}
```
Two threads operating on separate partitions

Kernel Function:

```c
void kern (int start, int end) {
    int i;
    for(i=start; i<=end; ++i) {
    }
}
```

Memory

```
A
1   n
```

Thread 0

```c
main() {
    ...
    kern (1, n/2);
    ...
}
```

Thread 1

```c
kern (n/2+1, n);
```
Recall: Amdahl’s law for theoretical speedup

Overall speedup is limited to the fraction of the program that can be executed in parallel

\[ \text{speedup} = \frac{1}{f + \frac{1-f}{n}} \]

\( f \): sequential fraction

Speedup vs. Sequential Fraction

- 10x
- 5x
- ~2x
- ~1x

Number of Processors

Speedup

10%  20%  40%  60%  90%
Example Code II

- A single location is updated every time

Kernel Function:
```c
void kern (int start, int end) {
    int i;
    for(i=start; i<=end; ++i) {
        sum = sum + A[i];
    }
}
```

Memory:
```c
int A[1...n]
```

Thread 0
```c
main() {
    ...
    kern (1, n);
    ...
}
```
A single location is updated every time

Kernel Function:
```c
void kern (int start, int end) {
    int i;
    for(i=start; i<=end; ++i) {
        sum = sum + A[i];
    }
}
```

Memory:
```
A
1
```

Thread 0
```
main() {
    ...
    kern (1, n);
    ...
}
```
Two threads operating on separate partitions

Kernel Function:

```c
void kern (int start, int end) {
    int i;
    for (i=start; i<=end; ++i) {
        sum = sum + A[i];
    }
}
```

```c
main() {
    ...
    kern (1, n/2);
    ...
}
```