DATA LEVEL PARALLELISM

Mahdi Nazm Bojnordi

Assistant Professor
School of Computing
University of Utah
Overview

- Announcement
  - Homework 5: due on Nov. 20th

- This lecture
  - Data level parallelism
Overview

- ILP: instruction level parallelism
  - Out of order execution (all in hardware)
  - IPC hardly achieves more than 2

- Other forms of parallelism
  - DLP: data level parallelism
    - Vector processors, SIMD, and GPUs
  - TLP: thread level parallelism
    - Multiprocessors, and hardware multithreading
  - RLP: request level parallelism
    - Datacenters
Data Level Parallelism (DLP)
Data Level Parallelism

- Due to executing the same code on a large number of objects
  - Common in scientific computing
- DLP architectures
  - Vector processors—e.g., Cray machines
  - SIMD extensions—e.g., Intel MMX
  - Graphics processing unit—e.g., NVIDIA
- Improve throughput rather than latency
  - Not good for non-parallel workloads
Vector Processing

- Scalar vs. vector processor

```c
for(i=0; i<1000; ++i) {
    B[i] = A[i] + x;
}
```

A:

```
... ...
```

B:

```
... ...
```
Scalar vs. vector processor

```c
for(i=0; i<1000; ++i) {
    B[i] = A[i] + x;
}
```

Add r3, r2, r1
Vector Processing

- Scalar vs. vector processor

```c
for(i=0; i<1000; ++i) {
    B[i] = A[i] + x;
}
```

A: ...

B: ...

vadd v3, v2, v1
Vector Processor

- A scalar processor—e.g., MIPS
  - Scalar register file
  - Scalar functional units

- Vector register file
  - 2D register array
  - Each register is an array of registers
  - The number of elements per register determines the max vector length

- Vector functional units
  - Single opcode activates multiple units
  - Integer, floating point, load and stores
Basic Vector Processor Architecture
Parallel vs. Pipeline Units
Vector Instruction Set Architecture

- Single instruction defines multiple operations
  - Lower instruction fetch/decode/issue cost
- Operations are executed in parallel
  - Naturally no dependency among data elements
  - Simple hardware
- Predictable memory access pattern
  - Improve performance via prefetching
  - Simple memory scheduling policy
  - Multi banking may be used for improving bandwidth
Vector Operation Length

- Fixed in hardware
  - Common in narrow SIMD
  - Not efficient for wide SIMD

- Variable length
  - Determined by a vector length register (VLR)
  - MVL is the maximum VL
  - How to process vectors wider than MVL?
Conditional Execution

- **Question:** how to handle branches?
- **Solution:** by predication
  - Use masks, flag vectors with single-bit elements
  - Determine the flag values based on vector compare
  - Use flag registers as control mask for the next vector operations

```c
for(i=0; i<1000; ++i) {
    if(A[i] != B[i])
        A[i] = B[i];
}
```

```c
vld V1, Ra
vld V2, Rb
vcmp.neq.vv M0, V1, V2
vsub.vv V3, V2, V1, M0
vst V3, Ra
```
Branches in Scalar Processors

```c
for (i = 0; i < 8; ++i) {
    if (inp[i] > 0) {
        y = inp[i] * inp[i];
        y = y + 2 * inp[i];
        out[i] = y + 3;
    } else {
        y = 4 * inp[i];
        out[i] = y + 1;
    }
}
```
for (i = 0; i < 8; ++i) {
    if (inp[i] > 0) {
        y = inp[i] * inp[i];
        y = y + 2 * inp[i];
        out[i] = y + 3;
    } else {
        y = 4 * inp[i];
        out[i] = y + 1;
    }
}
Branches in Scalar Processors

```c
for (i = 0; i < 8; ++i) {
    if (inp[i] > 0) {
        y = inp[i] * inp[i];
        y = y + 2 * inp[i];
        out[i] = y + 3;
    } else {
        y = 4 * inp[i];
        out[i] = y + 1;
    }
}
```
Branches in Vector Processors

```plaintext
if (inp[i] > 0) {
    y = inp[i] * inp[i];
    y = y + 2 * inp[i];
    out[i] = y + 3;
} else {
    y = 4 * inp[i];
    out[i] = y + 1;
}
```
Branches in Vector Processors

if (inp[i] > 0) {
    y = inp[i] * inp[i];
    y = y + 2 * inp[i];
    out[i] = y + 3;
} else {
    y = 4 * inp[i];
    out[i] = y + 1;
}
Branches in Vector Processors

```c
if (inp[i] > 0) {
    y = inp[i] * inp[i];
y = y + 2 * inp[i];
out[i] = y + 3;
}
else {
    y = 4 * inp[i];
out[i] = y + 1;
}
```
Graphics Processing Unit (GPU)
Graphics Processing Unit

- Initially developed as graphics accelerator
  - It receives geometry information from the CPU as an input and provides a picture as an output
Graphics Processing Unit

- Initially developed as graphics accelerator
  - It receives geometry information from the CPU as an input and provides a picture as an output
The host interface is the communication bridge between the CPU and the GPU. It receives commands from the CPU and also pulls geometry information from system memory. It outputs a stream of vertices in object space with all their associated information.
Vertex Processing

- The vertex processing stage receives vertices from the host interface in object space and outputs them in screen space.
- This may be a simple linear transformation, or a complex operation involving morphing effects.
Pixel Processing

- Rasterize triangles to pixels
- Each fragment provided by triangle setup is fed into fragment processing as a set of attributes (position, normal, texcoord etc), which are used to compute the final color for this pixel
- The computations taking place here include texture mapping and math operations
Programming GPUs

- The programmer can write programs that are executed for every vertex as well as for every fragment.
- This allows fully customizable geometry and **shading** effects that go well beyond the generic look and feel of older 3D applications.
Programming GPUs

- The programmer can write programs that are executed for every vertex as well as for every fragment.
- This allows fully customizable geometry and shading effects that go well beyond the generic look and feel of older 3D applications.
Memory Interface

- Fragment colors provided by the previous stage are written to the framebuffer.
- Used to be the biggest bottleneck before fragment processing took over.
- Before the final write occurs, some fragments are rejected by the zbuffer, stencil and alpha tests.
- On modern GPUs, z and color are compressed to reduce framebuffer bandwidth (but not size).
Z-Buffer

- Example of 3 objects
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
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

Diagram:
- Application Code
  - Compute-Intensive Functions
  - Use GPU to Parallelize
  - Rest of Sequential CPU Code

GPU

CPU
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()`, `cudaMemcpys()`, 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[]
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();
```
Example: Allocate Memory

```c
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