Parallel Architectures

Chris Rossbach and Calvin Lin
cs380p
Outline

Over the next few classes:

Background from many areas
  - Architecture
    - Vector processors
    - Hardware multi-threading
  - Graphics
    - Graphics pipeline
    - Graphics programming models
  - Algorithms
    - parallel architectures $\rightarrow$ parallel algorithms

Programming GPUs
  - CUDA
    - Basics: getting something working
    - Advanced: making it perform
Outline

Over the next few classes:

Background from many areas

This lecture

Algorithms
  parallel architectures → parallel algorithms

Programming GPUs
  CUDA
    Basics: getting something working
    Advanced: making it perform
A Modern GPU
A Modern GPU
A Modern GPU
A Modern GPU

80 SMs
A Modern GPU

- 80 SMs
- 64 cores/SM
- 5210 threads!
- 15.7 TFLOPS
A Modern GPU

- 80 SMs
- 64 cores/SM
- 5210 threads!
- 15.7 TFLOPS

All super-computers in the world!
A Modern GPU

80 SMs
64 cores/SM
5210 threads!
15.7 TFLOPS
A Modern GPU

- 80 SMs
- 64 cores/SIM
- 5210 threads!
- 15.7 TFLOPS
- 640 Tensor cores
A Modern GPU

- 80 SMs
- 64 cores/SM
- 5210 threads!
- 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
- 4096-bit bus
- No cache coherence!
A Modern GPU

- 80 SMs
- 64 cores/SM
- 5210 threads!
- 15.7 TFLOPS
- 640 Tensor cores
- HBM2 memory
- 4096-bit bus
- No cache coherence!
- 16 GB memory
- PCIe-attached
Chapter 1

Understanding the machine
Architecture Review: Pipelines

Processor algorithm:

```c
main() {
    while(true) {
        do_next_instruction();
    }
}
```
Processor algorithm:

```c
main() {
    while(true) {
        do_next_instruction();
    }
}
```

```c
do_next_instruction() {
    instruction = fetch();
    ops, regs = decode(instruction);
    execute_calc_addr(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}
```
Architecture Review: Pipelines

Processor algorithm:
```java
main() {
    while(true) {
        do_next_instruction();
    }
}
```

```java
do_next_instruction() {
    instruction = fetch();
    ops, regs = decode(instruction);
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}
```
Architecture Review: Pipelines

Processor algorithm:

```c
main() {
    while(true) {
        do_next_instruction();
    }
}
```

```c
do_next_instruction() {
    instruction = fetch();
    ops, regs = decode(instruction);
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}
```
Architecture Review: Pipelines

Processor algorithm:
```
main() {
    while(true) {
        do_next_instruction();
    }
}
```

```
do_next_instruction() {
    instruction = fetch();
    ops, regs = decode(instruction);
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}
```

Works well if pipeline is kept full
What kinds of things cause “bubbles”/stalls?
Processor algorithm:

```java
main() {
    while(true) {
        do_next_instruction();
    }
}
```

do_next_instruction() {
    instruction = fetch();
    ops, regs = decode(instruction);
    execute_calc_addr(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}
Multi-core/SMPs
Multi-core/SMPs
Multi-core/SMPs
Multi-core/SMPs
main() {
    for (i=0; i<CORES; i++) {
        pthread_create(
            do_next_instruction);
    }
}

do_next_instruction() {
    instruction = fetch();
    ops, regs = decode(instruction);
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}
Multi-core/SMPs

```c
main() {
    for(i=0; i<CORES; i++) {
        pthread_create(
            do_next_instruction());
    }
}

do_next_instruction() {
    instruction = fetch();
    ops, regs = decode(instruction);
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}
```

- **Pros: Simple**
- **Cons: programmer has to find the parallelism!**
main() {
    for(i=0; i<CORES; i++) {
        pthread_create(
            do_next_instruction();
        }
    }
}

Other techniques extract parallelism here, try to let the machine find parallelism!
Superscalar processors
Superscalar processors
Superscalar processors
Superscalar processors
Superscalar processors

```c
main() {
    for(i=0; i<CORES; i++)
        pthread_create(decode_exec);
    while(true) {
        instruction = fetch();
        enqueue(instruction);
    }
}

decode_exec() {
    instruction = dequeue();
    ops, regs = decode(instruction);
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}
```
Superscalar processors

main() {
  for(i=0; i<CORES; i++)
    pthread_create(decode_exec);
  while(true) {
    instruction = fetch();
    enqueue(instruction);
  }
}

decode_exec() {
  instruction = dequeue();
  ops, regs = decode(instruction);
  execute_calc_addrs(ops, regs);
  access_memory(ops, regs);
  write_back(regs);
}

Doesn’t look that different does it? Why do it?
Superscalar processors

main() {
    for(i=0; i<CORES; i++)
        pthread_create(decode_exec);
    while(true) {
        instruction = fetch();
        enqueue(instruction);
    }
}

decode_exec() {
    instruction = dequeue();
    ops, regs = decode(instruction);
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}

Doesn’t look that different does it? Why do it?

Enables independent instruction parallelism.
Superscalar processors

main() {
    for(i=0; i<CORES; i++)
        pthread_create(decode_exec);
    while(true) {
        instruction = fetch();
        enqueue(instruction);
    }
}

decode_exec() {
    instruction = dequeue();
    ops, regs = decode(instruction);
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}

Doesn’t look that different does it? Why do it?

Enables independent instruction parallelism.
Superscalar processors

main() {
    for(i=0; i<CORES; i++)
        pthread_create(decode_exec);
    while(true) {
        instruction = fetch();
        enqueue(instruction);
    }
}

decode_exec() {
    instruction = dequeue();
    ops, regs = decode(instruction);
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}

Doesn’t look that different does it? Why do it?

Enables independent instruction parallelism.
Vector/SIMD processors

```c
#include <stdio.h>

int main(void)
{
    int i, j;
    float A[64], B[64], C[64];

    for (i = 0; i < 64; i++)
        C[i] = A[i] + B[i];

    // C code
    for (i = 0; i < 64; i++)
        C[i] = A[i] + B[i];

    // Scalar Code
    for (i = 0; i < 64; i++)
        C[i] = A[i] + B[i];
```
Vector/SIMD processors

```c
# C code
for (i=0; i<64; i++)
C[i] = A[i] + B[i];
```

```c
# Scalar Code
LI    R4, 64
loop:
L.D   F0, 0(R1)
L.D   F2, 0(R2)
ADD.D F4, F2, F0
S.D   F4, 0(R3)
DADDIU R1, 8
DADDIU R2, 8
DADDIU R3, 8
DSUBIU R4, 1
BNEZ   R4, loop
```
Vector/SIMD processors

Why decode same instruction over and over?
Vector/SIMD processors
Vector/SIMD processors

main() {
    for(i=0; i<CORES; i++)
        pthread_create(exec);
    while(true) {
        ops, regs = fetch_decode();
        enqueue(ops, regs);
    }
}

eexec() {
    ops, regs = dequeue();
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}
main() {
    for(i=0; i<CORES; i++)
        pthread_create(exec);
    while(true) {
        ops, regs = fetch_decode();
        enqueue(ops, regs);
    }
}

exec() {
    ops, regs = dequeue();
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}
Vector/SIMD processors

main() {
    for (i=0; i<CORES; i++)
        pthread_create(exec);
    while (true) {
        ops, regs = fetch_decode();
        enqueue(ops, regs);
    }
}

exec() {
    ops, regs = dequeue();
    execute_calc_addrs(ops, regs);
    access_memory(ops, regs);
    write_back(regs);
}
Vector Processors

Process multiple data elements simultaneously.

Common in supercomputers of the 1970’s 80’s and 90’s.

Modern CPUs support some vector processing instructions
   Usually called SIMD

Can operate on few vector elements per clock cycle in a pipeline or,
   SIMD operate on all per clock cycle
Vector Processors

Process multiple data elements simultaneously.

Common in supercomputers of the 1970’s 80’s and 90’s.

Modern CPUs support some vector processing instructions

Usually called SIMD

Can operate on few vector elements per clock cycle in a pipeline or,

SIMD operate on all per clock cycle

- 1962 University of Illinois Illiac IV - completed 1972 → 64 ALUs 100-150 MFlops
- (1973) TI’s Advance Scientific Computer (ASC) 20-80 MFlops
- (1975) Cray-1 first to have vector registers instead of keeping data in memory
Vector Processors

Process multiple data elements simultaneously.

Common in supercomputers of the 1970’s 80’s and 90’s.

Modern CPUs support some vector processing instructions
    Usually called SIMD

Can operate on few vector elements per clock cycle in a pipeline or,
    SIMD operate on all per clock cycle

- 1962 University of Illinois Illiac IV - completed 1972 → 64 ALUs 100-150 MFlops
- (1973) TI’s Advance Scientific Computer (ASC) 20-80 MFlops
- (1975) Cray-1 first to have vector registers instead of keeping data in memory
When does vector processing help?
When does vector processing help?

What are the potential bottlenecks here?
When can it improve throughput?
When does vector processing help?

What are the potential bottlenecks here?
When can it improve throughput?

Only helps if memory can keep the pipeline busy!
HW multi-threading
HW multi-threading

Address memory bottleneck
HW multi-threading

Address memory bottleneck
Share exec unit across
  Instruction streams
  Switch on stalls
HW multi-threading

Address memory bottleneck
Share exec unit across
  Instruction streams
  Switch on stalls
HW multi-threading

Address memory bottleneck
Share exec unit across
Instruction streams
Switch on stalls
HW multi-threading

Address memory bottleneck
Share exec unit across
  Instruction streams
  Switch on stalls
Looks like multiple cores to the OS
HW multi-threading

Address memory bottleneck
Share exec unit across
  Instruction streams
  Switch on stalls
Looks like multiple cores to the OS
Three variants:
  Coarse
  Fine-grain
  Simultaneous
### Running Example

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread C</th>
<th>Thread D</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Colors" /></td>
<td><img src="#" alt="Colors" /></td>
<td><img src="#" alt="Colors" /></td>
<td><img src="#" alt="Colors" /></td>
</tr>
</tbody>
</table>

- Colors → pipeline full
- White → stall
Coarse-grain Multi-threading
Coarse-grain Multi-threading

Single thread run until costly stall
  e.g. 2nd level cache miss
Coarse-grain Multi-threading

Single thread run until costly stall
  e.g. 2nd level cache miss

Another thread starts during stall
  Pipeline fill time requires several cycles!
Coarse-grain Multi-threading

Single thread run until costly stall
  e.g. 2nd level cache miss

Another thread starts during stall
  Pipeline fill time requires several cycles!

Does not cover short stalls
Coarse-grain Multi-threading

Single thread run until costly stall
e.g. 2nd level cache miss

Another thread starts during stall
   Pipeline fill time requires several cycles!

Does not cover short stalls

Hardware support required
   PC and register file for each thread
   little other hardware
   Looks like another CPU to OS
Coarse-grain Multi-threading

Single thread run until costly stall
e.g. 2nd level cache miss

Another thread starts during stall
Pipeline fill time requires several cycles!

Does not cover short stalls

Hardware support required
PC and register file for each thread
little other hardware
Looks like another CPU to OS
Coarse-grain Multi-threading

Single thread run until costly stall
e.g. 2nd level cache miss

Another thread starts during stall
Pipeline fill time requires several cycles!

Does not cover short stalls

Hardware support required
PC and register file for each thread
little other hardware
Looks like another CPU to OS
Fine-grained multithreading
Fine-grained multithreading

Two+ threads interleave instructions
  Round-robin fashion
  Skip stalled threads
Fine-grained multithreading

Two+ threads interleave instructions
  Round-robin fashion
  Skip stalled threads

Hardware support required
  Separate PC and register file for each thread
  Hardware to control alternating pattern
Fine-grained multithreading

Two+ threads interleave instructions
  Round-robin fashion
  Skip stalled threads

Hardware support required
  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
Fine-grained multithreading

Two+ threads interleave instructions
  Round-robin fashion
  Skip stalled threads

Hardware support required
  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

Two+ threads interleave instructions
- Round-robin fashion
- Skip stalled threads

Hardware support required
- 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
Simultaneous Multithreading (SMT)
Simultaneous Multithreading (SMT)

Instructions from multiple threads issued per cycle

- Uses register renaming
- Dynamic scheduling facility of multi-issue architecture
Simultaneous Multithreading (SMT)

Instructions from multiple threads issued per cycle
  Uses register renaming
  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
Simultaneous Multithreading (SMT)

Instructions from multiple threads issued per cycle
  Uses register renaming
  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 (SMT)

Instructions from multiple threads issued per cycle
  Uses register renaming
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
Why Vector and MT Background?

GPU:
• A very wide vector machine
• Massively multi-threaded to hide memory latency
• Originally designed for graphics pipelines...
Graphics = Rendering

Inputs:

3D world model (objects, materials)
- Geometry modeled using triangle meshes + surface normals
- GPUs subdivide triangles into “fragments” (rasterization)
- Materials modeled with “textures”
- Texture coordinates and sampling to map textures → geometry

Light locations and properties
- Attempt to model surface/light interactions with modeled objects/materials

View point

Output:

2D projection seen from the view-point
Simplified Rendering Algorithm

foreach(vertex v in model)
    map v_{model} \rightarrow v_{view}

fragment[] frags = {};

foreach triangle t (v_0, v_1, v_2)
    frags.add(rasterize(t));

foreach fragment f in frags
    choose_color(f);

display(visible_fragments(frags));
Algorithm → Graphics Pipeline

```java
foreach(vertex v in model)
    map v_{model} → v_{view}
fragment[] frags = {};
foreach triangle t (v_0, v_1, v_2)
    frags.add(rasterize(t));
foreach fragment f in frags
    choose_color(f);
display(visible_fragments(frags));
```

OpenGL pipeline

To first order, DirectX looks the same!
foreach(vertex v in model)

fragment[] frags = {};
foreach triangle t (v₀, v₁, v₂)
  frags.add(rasterize(t));
foreach fragment f in frags
  choose_color(f);
display(visible_fragments(frags));

OpenGL pipeline
To first order, DirectX looks the same!
Algorithm ➔ Graphics Pipeline

```plaintext
define  
  foreach(vertex v in model) 
    map v_{model} ➔ v_{view} 
fragment[] frags = {}; 
foreach triangle t (v_0, v_1, v_2) 
  frags.add(rasterize(t)); 
foreach fragment f in frags 
  choose_color(f); 
  display(visible_fragments(frags)); 
```

OpenGL pipeline

To first order, DirectX looks the same!
Algorithm → Graphics Pipeline

foreach(vertex v in model)
    map v_{model} \rightarrow v_{view}
fragment[] frags = {};
foreach triangle t (v_0, v_1, v_2)
    frags.add(rasterize(t));
foreach fragment f in frags
display(visible_fragments(frags));

OpenGL pipeline

To first order, DirectX looks the same!
Algorithm → Graphics Pipeline

```plaintext
foreach(vertex v in model)
    map \( v_{\text{model}} \rightarrow v_{\text{view}} \)
fragment[] frags = {};
foreach triangle t (v\(_0\), v\(_1\), v\(_2\))
    frags.add(rasterize(t));
foreach fragment f in frags
    choose_color(f);
```

To first order, DirectX looks the same!
Limited “programmability” of shaders:
Minimal/no control flow
Maximum instruction count

GeForce 6 series
Limited “programmability” of shaders:
Minimal/no control flow
Maximum instruction count

GeForce 6 series
Limited “programmability” of shaders:
- Minimal/no control flow
- Maximum instruction count

GeForce 6 series
Limited “programmability” of shaders:
Minimal/no control flow
Maximum instruction count

GeForce 6 series
Limited “programmability” of shaders:
Minimal/no control flow
Maximum instruction count

Graphics pipeline → GPU architecture

GeForce 6 series
Late Modernity: unified shaders

Mapping to Graphics pipeline no longer apparent
Processing elements no longer specialized to a particular role
Model supports *real* control flow, larger instr count
Modern: Pascal
Pascal SM
Cross-generational observations

GPUs designed for parallelism in graphics pipeline:

Data
- Per-vertex
- Per-fragment
- Per-pixel

Task
- Vertex processing
- Fragment processing
- Rasterization
- Hidden-surface elimination

MLP
- HW multi-threading for hiding memory latency
Cross-generational observations

GPUs designed for parallelism in graphics pipeline:

Data
- Per-vertex
- Per-fragment
- Per-pixel

Task
- Vertex processing
- Fragment processing
- Rasterization
- Hidden-surface elimination

MLP
- HW multi-threading for hiding memory latency

Even as GPU architectures become more general, certain assumptions persist:
1. Data parallelism is *trivially* exposed
2. All problems look like painting a box with colored dots
Cross-generational observations

GPUs designed for parallelism in graphics pipeline:

**Data**
- Per-vertex
- Per-fragment
- Per-pixel

**Task**
- Vertex processing
- Fragment processing
- Rasterization
- Hidden-surface elimination

Even as GPU architectures become more general, certain assumptions persist:

1. Data parallelism is *trivially* exposed
2. All problems look like painting a box with colored dots

**MLP**
- HW multi-threading for hiding memory latency

*But what if my workload isn’t “painting a box”?!!?!!*
Summary

Key Ideas:
Simple cores
Single instruction stream
  Vector instructions (SIMD) OR
  Implicit HW-managed sharing (SIMT)
Hide memory latency with HW multi-threading