Introduction

Trend toward parallel workloads led to need for implicit instruction level parallelism from a single thread on CPU.

This made the job of the architect very difficult trying to design around complex instruction scheduling logic.

GPGPU programming has moved toward solving this problem through explicit thread level parallelism.

The software developer now has to do the work in their programming model.
Introduction

Exploiting explicit thread level parallelism is achieved through the SIMD programming model.

In the Nvidia programming language CUDA, threads are grouped in SIMD warps.

The SIMD warps are scheduled to execute based on their program counter.

If the threads in the SIMD warp have different PCs as a result from different decisions on a branch, the warp will encounter branch divergence.

SIMD instructions should execute in lockstep, if divergence is encountered, the execution in a warp should be forced to serialize.
SIMD Stream Processor Architecture

SIMD - Single Instruction Multi Data

Exploit parallelism in a single instruction by packing vector operations into a single instruction (Dot Product becomes a single ADD X1 X2 instruction)

In the GPGPU SIMD model, a warp operates on a single instruction and each thread in a warp operates on an individual piece of data
**Latency Hiding**

Requirement to hide latency from memory access time. We do not want to stall all other instructions on memory requests.

When a thread makes a request to memory, the blocking thread is added to a fair round robin queue to be scheduled when it is ready to continue.

The next warp is now scheduled, effectively giving us out of order warp execution to hide the memory latency incurred from a memory access.

This latency hiding is called *barrel processing*.
Latency Hiding

Dynamic Warp Formation and Scheduling for GPU Control Flow
SIMD Execution of Scalar Threads

The GPGPU programming model is supported by the GPU hardware inorder to achieve performance increases from explicit parallelism.

The SIMD warp is spread across multiple scalar pipelines to be executed in “lock-step”.

The SIMD scheduler will only schedule threads in a warp with the same PC. As divergence occurs, the SIMD scheduler serializes divergent threads.

Grouping scalar threads into a SIMD warp
**SIMD Control Flow Support**

Predication is a natural way for programs to have fine-grained control flow on the SIMD pipeline.

Predication does not eliminate branches, therefore, we still have issue of branch divergence.

The SIMD pipeline is fully utilized when executing all threads in a warp in “lock-step”.

Therefore, in software containing many branching instructions, it is lucrative to mitigate the performance latency incurred from branch divergence.
SIMD Serialization

Naive approach is to serialize branching instructions. In the worst case, our warp performs as a SISD pipeline serializing n threads in the warp

SIMD Serialization loses the performance increase we gain from executing threads in parallel. If we wanted a serialized pipeline, it would be better to execute on CPU

Each node represents a PC. This shows worst case serialization, where all threads in a warp of 8 threads end up at different PCs
SIMD Reconvergence

Definitions:

**Immediate Post-Dominator** - reconvergence point of a diverging branch

**Post-Dominator** - A basic block X post-dominates basic block Y (x pdom y), iff all paths from y to the exit node go through X, where a basic block is a piece of code with a single entry and exit point

**Immediately Post-Dominates** - A basic block X, distinct from Y, immediately post-dominates basic block Y iff X pdom Y and there is no basic block Z such that X pdom Z and Z pdom Y
Post Dominator Example

E immediately post dominates B

G post dominates B

G immediately post dominates A
**SIMD Reconvergence**

Reconverging control flow can decrease the number of threads in the warp that we must serialize.

We now group diverging threads based on equivalent PCs.

We can increase warp utilization, but we are still forced to serialize diverging threads.

We can still have worst case $n$ diverging threads that we would be forced to serialize in a warp.
An example for PDOM
An example for PDOM
An example for PDOM
An example for PDOM
An example for PDOM
An example for PDOM

Stack

<table>
<thead>
<tr>
<th>Reconv. PC</th>
<th>Next PC</th>
<th>Active Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>E</td>
<td>1111</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
<td>0110</td>
</tr>
<tr>
<td>E</td>
<td>C</td>
<td>1001</td>
</tr>
</tbody>
</table>

Time

A/1111

B/1111

C/1001

D/0110

F

G

.......

A       B       C
An example for PDOM

Stack

<table>
<thead>
<tr>
<th>Reconv. PC</th>
<th>Next PC</th>
<th>Active Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>E</td>
<td>1111</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
<td>0110</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>1001</td>
</tr>
</tbody>
</table>
An example for PDOM

Stack

<table>
<thead>
<tr>
<th>Reconv. PC</th>
<th>Next PC</th>
<th>Active Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>E</td>
<td>1111</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
<td>0110</td>
</tr>
</tbody>
</table>

Time
An example for PDOM
An example for PDOM
An example for PDOM
PDOM Performance

![Bar Chart: IPC Normalized to Theoretical Peak Machine Parallelism]

- 4 Wide: 0.8
- 8 Wide: 0.6
- 16 Wide: 0.4
A Counterexample to PDOM

```c
void shader_thread(int tid, int *data) {
    for(int i = tid % 2; i < 128; ++i) {
        if(i % 2) {
            data[tid]++;
        }
    }
}
```
Question

What is parallel iterative matching allocator?

What exactly is the Needleman-Wunsch algorithm doing, as mentioned on page 411?
Dynamic Warp Formation and Scheduling

Image from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt
Dynamic Warp Formation and Scheduling

Image from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt
Register File Access

- We're assuming that all RFs are equally accessible by all lanes.
- This is problematic...

Figure 10. Register file configuration for (a) ideal dynamic warp formation and MIMD, (b) naïve dynamic warp formation, (c) static warp formation, (d) lane-aware dynamic warp formation.
Register File Access

- In reality, each lane has its own register file bank with data only accessible within that lane.
- How do we avoid shuttling register values around?

Figure 10. Register file configuration for (a) ideal dynamic warp formation and MIMD, (b) naïve dynamic warp formation, (c) static warp formation, (d) lane-aware dynamic warp formation.
Register File Access

- Figure 10(c)
  - static warp formation
  - Depicts a warp of threads accessing their RFs
  - Each vertical RF/ALU pair is a lane for a thread
  - Depicts each lane’s copy of the desired register being accessed

Figure 10. Register file configuration for (a) ideal dynamic warp formation and MIMD, (b) naïve dynamic warp formation, (c) static warp formation, (d) lane-aware dynamic warp formation.
Register File Access

- **Figure 10(b)**
  - naïve dynamic warp formation with no regard for home lanes
  - Bank conflicts are possible
- A **crossbar** is needed for when threads leave their home lane. The crossbar remaps RF banks such that the appropriate bank is available to a thread.

![Register File Access Diagram](image)

Figure 10. Register file configuration for (a) ideal dynamic warp formation and MIMD, (b) naïve dynamic warp formation, (c) static warp formation, (d) lane-aware dynamic warp formation.
Q: In 4.1, register file crossbars are mentioned. Could you explain what the purpose of the crossbar is and how the authors fix the 'drawbacks' of this with the "lane aware dynamic warp formation"?
Q: In 4.1, register file crossbars are mentioned. Could you explain what the purpose of the crossbar is and how the authors fix the ‘drawbacks’ of this with the “lane aware dynamic warp formation”?

A: In this case, the crossbar is a hardware structure that allows one lane to access another lane’s register file bank. The authors propose “lane aware” warp formation to avoid the need for a crossbar. See Hardware slides for an illustrative example.
Register File Access

- Figure 10(c)
  - lane aware dynamic warp formation
- If we force threads to stay in their home lanes, we don’t need a crossbar!

Figure 10. Register file configuration for (a) ideal dynamic warp formation and MIMD, (b) naïve dynamic warp formation, (c) static warp formation, (d) lane-aware dynamic warp formation.
Hardware Implementation

Animation and Images from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt
Hardware Implementation

Thread Scheduler
- Warp Update Register T
  - TID x N, REQ, PC A
- Warp Update Register NT
  - TID x N, REQ, PC B

PC-Warp LUT
- PC, OCC, IDX

Warp Pool
- PC, TID x N, Prio

Warp Allocator

Issue Logic

Committed/Writeback

Animation and images from “Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow”, Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt.
Hardware Implementation

Animation and Images from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt
Hardware Implementation

Thread Scheduler

Warp Update Register T
- TID x N
- REQ
- PC A

Warp Update Register NT
- TID x N
- REQ
- PC B

PC-Warp LUT
- PC
- OCC
- IDX

Warp Pool
- PC
- TID x N
- Prio

Warp Allocator

Issue Logic

A: BEQ R2, B
C: ...

Animation and Images from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt.
Hardware Implementation

Thread Scheduler

Warp Update Register T
TID x N  REQ  PC A

Warp Update Register NT
TID x N  REQ  PC B

PC-Warp LUT
PC  OCC  IDX

Warp Pool
PC  TID x N  Prio
PC  TID x N  Prio
PC  TID x N  Prio

Warp Allocator

Issue Logic

Animation and Images from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt.
Hardware Implementation

Thread Scheduler

Warp Update Register T

TID x N | REQ | PC A

Warp Update Register NT

TID x N | REQ | PC B

PC-Warp LUT

PC | OCC | IDX

Warp Pool

PC | TID x N | Prio

Warp Allocator

X

1

2

3

4

(ALU 1, ID)

(ALU 2, ID)

(ALU 3, ID)

(ALU 4, ID)

Y

5

6

7

8

(I-Cache)

A: BEQ R2, B

C: …
Hardware Implementation

Thread Scheduler
Warp Update Register T
TID x N | REQ | PC A
Warp Update Register NT
TID x N | REQ | PC B

PC-Warp LUT
PC | OCC | IDX

Warp Pool
PC | TID x N | Prio

Issue Logic

Warp Allocator
X
Y

Animation and Images from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt
Hardware Implementation

Thread Scheduler

Warp Update Register T
- TID x N
- REQ
- PC A

Warp Update Register NT
- TID x N
- REQ
- PC B

PC-Warp LUT
- PC
- OCC
- IDX

Warp Pool
- PC
- TID x N
- Prio

Warp Allocator

Issue Logic

Animation and images from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt.
Hardware Implementation

Thread Scheduler

Warp Update Register T

Warp Update Register NT

PC-Warp LUT

Warp Pool

Warp Allocator

Issue Logic

Animation and Images from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt.
Hardware Implementation

Thread Scheduler

Warp Update Register T

| TID x N | 2 | 3 | 0110 | B |

Warp Update Register NT

| TID x N | 1 | 4 | 1001 | C |

PC-Warp LUT

<table>
<thead>
<tr>
<th>B</th>
<th>0110</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>OCC</td>
<td>IDX</td>
</tr>
</tbody>
</table>

Warp Pool

<table>
<thead>
<tr>
<th>B</th>
<th>TID x N</th>
<th>Prio</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>TID x N</td>
<td>Prio</td>
</tr>
</tbody>
</table>

Warp Allocator

Issue Logic

ALU 1

ALU 2

ALU 3

ALU 4

RF 1

RF 2

RF 3

RF 4

Commit/Writeback

Animation and Images from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt.

A: BEQ R2, B

C: ...
Hardware Implementation

Thread Scheduler

Warp Update Register T

<table>
<thead>
<tr>
<th>ID x N</th>
<th>0110</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Warp Update Register NT

<table>
<thead>
<tr>
<th>ID x N</th>
<th>1001</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PC-Warp LUT

<table>
<thead>
<tr>
<th>B</th>
<th>0110</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1001</td>
<td>1</td>
</tr>
</tbody>
</table>

Warp Pool

<table>
<thead>
<tr>
<th>B</th>
<th>TID x N</th>
<th>Prio</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Warp Allocator

<table>
<thead>
<tr>
<th>PC</th>
<th>TID x N</th>
<th>Prio</th>
</tr>
</thead>
</table>

Issue Logic

Animation and Images from “Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow”, Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt.
Hardware Implementation

Thread Scheduler

Warp Update Register T

| TID x N | 0100 | C |

Warp Update Register NT

| TID x N | 0100 | C |

PC-Warp LUT

<table>
<thead>
<tr>
<th>B</th>
<th>0110</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1001</td>
<td>1</td>
</tr>
</tbody>
</table>

Warp Pool

<table>
<thead>
<tr>
<th>B</th>
<th>TID x N</th>
<th>Prio</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Warp Allocator

| TID x N | Prio |

Issue Logic

Animation and Images from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt.
Hardware Implementation

Thread Scheduler

Warp Update Register T

<table>
<thead>
<tr>
<th>TID</th>
<th>5</th>
<th>7</th>
<th>8</th>
<th>1011</th>
<th>B</th>
</tr>
</thead>
</table>

Warp Update Register NT

<table>
<thead>
<tr>
<th>TID x N</th>
<th>0100</th>
<th>C</th>
</tr>
</thead>
</table>

PC-Warp LUT

<table>
<thead>
<tr>
<th>PC</th>
<th>B 0010</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>C 1001</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Warp Pool

<table>
<thead>
<tr>
<th>TID x N</th>
<th>B 5 2 3 8</th>
<th>Prio</th>
</tr>
</thead>
<tbody>
<tr>
<td>TID x N</td>
<td>C 1 4</td>
<td>Prio</td>
</tr>
<tr>
<td>TID x N</td>
<td>B 7</td>
<td>Prio</td>
</tr>
</tbody>
</table>

Warp Allocator

Issue Logic

Commit/Writeback

ALU 1

RF 1

(TID, Reg#)

ALU 2

RF 2

(TID, Reg#)

ALU 3

RF 3

(TID, Reg#)

ALU 4

RF 4

(TID, Reg#)

Decode

I-Cache

Animation and images from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt.
Notice that threads stay in their home lanes. If we were to swap the position of any two threads, we’d need a crossbar.
Notice that we can’t add Thread 7 to warp index 0 because we’d have a bank conflict!
Hardware Implementation

Thread Scheduler

Warp Update Register T

TID 7 8 1011 B

Warp Update Register NT

TID x N 0100 C

PC-Warp LUT

B 0010 2

C 1001 1

Warp Pool

B 5 2 3 8 Prio

C 1 10 x N Prio

B TID 7 Prio

Issue Logic

Warp Allocator

ALU 1

ALU 2

ALU 3

ALU 4

RF 1

(TID, Reg#)

RF 2

(TID, Reg#)

RF 3

(TID, Reg#)

RF 4

(TID, Reg#)

I-Cache

Commit/Writeback

Animation and Images from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt.
Hardware Implementation

Thread Scheduler

Warp Update Register T

<table>
<thead>
<tr>
<th>TID x N</th>
<th>0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

Warp Update Register NT

<table>
<thead>
<tr>
<th>TID x N</th>
<th>0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

PC-Warp LUT

<table>
<thead>
<tr>
<th>PC</th>
<th>Warp</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0010</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1101</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Warp Pool

<table>
<thead>
<tr>
<th>TID</th>
<th>PC</th>
<th>Prio</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1101</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1101</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1101</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1101</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1101</td>
<td>1</td>
</tr>
</tbody>
</table>

Warp Allocator

Issue Logic

Animation and Images from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt.
Hardware Implementation

Thread Scheduler

Warp Update Register T

Warp Update Register NT

PC-Warp LUT

Warp Pool

Warp Allocator

Issue Logic

Animation and Images from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt.
Animation and images from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt.
Hardware Implementation

Thread Scheduler

Warp Update Register T

<table>
<thead>
<tr>
<th>TID</th>
<th>7</th>
<th>8</th>
<th>1011</th>
<th>B</th>
</tr>
</thead>
</table>

Warp Update Register NT

<table>
<thead>
<tr>
<th>TID x N</th>
<th>0100</th>
<th>C</th>
</tr>
</thead>
</table>

PC-Warp LUT

<table>
<thead>
<tr>
<th>PC</th>
<th>TID x N</th>
<th>Prio</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1101</td>
<td>1</td>
</tr>
</tbody>
</table>

Warp Pool

<table>
<thead>
<tr>
<th>PC</th>
<th>TID x N</th>
<th>Prio</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>TID x N</td>
<td>7</td>
</tr>
</tbody>
</table>

Warp Allocator

Issue Logic

Commit/Writeback

No Lane Conflict

Animation and Images from "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow", Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt.
Q: Section 4.2 "The warp with the older PC still resides in the warp pool, but will no longer be updated..." Does this mean that older warps just sit around indefinitely and accumulate in the warp pool?
A: No, when an old (full) warp is eventually issued, the data in the warp pool at that index must be invalidated. Once issued, the index (IDX value) is returned to the list of available indices for the Warp Allocator to use.
Issue Heuristics

- The issue priority is determined by the issue heuristic
- Majority Heuristic
  - Chooses the most common PC among all the existing warps and issues all before choosing a new PC
- More on these in the Performance section!

Figure 11. Implementation of dynamic warp formation and scheduling. In this figure, $H$ represents a hash operation. $N$ is the width of the SIMD pipeline.
Q: Section 4.3: "Each cycle, the issue logic searches for or allocates an entry for the PC of each warp entering the scheduler and increments the associated counter with the number of scalar threads joining the warp pool". What is this counter here and what is its purpose?
A: This is a description of the Issue Logic for the Majority Issue Heuristic (the authors use a 32 entry fully-associative LUT). Counters for each in-flight PC are used within this structure to keep track of which PC is the most common (i.e. which PC has “the Majority”) among all existing warps.
Area Estimation

- Overall area consumption is $2.799 \text{mm}^2$ per core
- With 8 cores, this is roughly 4.7% of the total area of the GeForce 8800GTX
- CACTI tool for estimation
Methodology

- **GPGPU-Sim**
  - Cycle-accurate simulator developed by the authors
  - Benchmarks included SPEC CPU2006, SPLASH2, and CUDA SDK Code Samples

- **Hardware configuration under test shown in Table 2**

<table>
<thead>
<tr>
<th>Table 2. Hardware Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td># Shader Cores</td>
</tr>
<tr>
<td>SIMD Warp Size</td>
</tr>
<tr>
<td># Threads per Shader Core</td>
</tr>
<tr>
<td># Memory Modules</td>
</tr>
<tr>
<td>GDDR3 Memory Timing</td>
</tr>
<tr>
<td>Bandwidth per Memory Module</td>
</tr>
<tr>
<td>Memory Controller</td>
</tr>
<tr>
<td>Data Cache Size (per core)</td>
</tr>
<tr>
<td>Data Cache Hit Latency</td>
</tr>
<tr>
<td>Default Warp Issue Heuristic</td>
</tr>
</tbody>
</table>
Experimental Results

- MIMD, being Multiple Insn Multiple Data, obviously has better performance than all the SIMD designs because it can execute different insns (with different PCs) in parallel.
- Naive - Normal SIMD with no reconvergence - lowest performance
- PDOM - reconvergence at post dominator- 93.4% speedup over naive
- DYNB (Majority)- 22.5% speedup over PDOM

Figure 12. Performance comparison of Naïve, PDOM, and DYNB versus MIMD.
Effects of Issue Heuristics

- Figure 15 shows SIMD performance across some benchmarks for different warp issue heuristics.
- In general, DPdPri, DPC and DMaj perform well.

Figure 15. Comparison of warp issue heuristics.
Effects of Issue Heuristics

- Figure 16 shows distribution of warps issued according to size.
- Say a warp has a capacity of 16 threads.
- W0 - 0 threads in warp
- W1 - 1 thread in warp
- W16 - 16 threads in warp - Full Warp!

More high occupancy warps issued - Good! Eg. DPC, DPdPri, DMaj
More low occupancy warps issued - Bad! Eg. DTime, DMin

STALL - No 2 threads can write to same register file in the same cycle, so writes are delayed
Q. What is the difference between stall and W0 cycles in Figure 16?

W0 means a warp with 0 threads >> No operations executed by the warp, more like a nop.

My guess is a scenario with __syncthreads(). Threads that have already reached this point must keep executing nops till the other threads come to this point.

Stall happens because of memory dependency. Multiple threads can’t write back to the same register bank in same cycle >> contention for same memory location. Hence, writes stall.

>> th1 can write 1st >> th2 can write >> 1 cycle of stall
Q. Could you explain in detail about Figure 16? I don't understand why certain warp, like W4 occupies a large portion?

W4 implies the warp has an occupancy of 4 threads. Occupying a large portion implies warps with occupancy 4 are more frequently issued. Means incomplete warps are being issued more often thereby >> under utilizing the SIMD pipeline.
Effect of Lane Aware Scheduling

- This figure compares PDOM vs DYNB with/without lane aware scheduling vs DYNB without lane conflict.
- As expected, DYNB with lane aware scheduling gives higher IPC than DYNB w/o lane aware (because of possible register bank conflict !)
- Ignoring lane conflicts gives even higher performance than the two.
- In case of Black, FFT and Matrix benchmarks however, PDOM gives better performance than DYNB.

*Figure 17. Performance of dynamic warp formation with lane aware scheduling and accounting for register file bank conflicts and scheduler implementation details.*
Related Work

- Predication
  - Execution of “predicated” insns is controlled by conditional mask set by another insn
  - Convert control dependency into data dependency

- Lorie and Strong
  - Introduce reconvergence point at the beginning of branch, rather than at the post-dominator
  - Eg. JOIN and ELSE instruction at the beginning of divergence

- Cervini
  - Dynamic regrouping of threads in a SPMD model on a SMT processor
  - After divergence, each thread has a single SIMD task

- Liquid SIMD (Clark et al.)
  - Form SIMD instructions by translating scalar instructions at runtime
  - Improves SIMD binary compatibility

- Conditional Routing (Kapasi)
  - Creates multiple kernels from single kernel to eliminate branches
  - Kernels connect via interconnect to increase SIMD pipeline utilization
Conclusion

- Branch divergence can significantly degrade a GPU’s performance. -50.5% performance loss with SIMD width = 16
- Dynamic Warp Formation & Scheduling -20.7% on average better than reconvergence -4.7% area cost
- Future Work -Warp scheduling – Area and Performance Tradeoff
Thank You !
Questions

1.) According to Wikipedia, SPMD is supposedly a subcategory of MIMD, so I'm confused by the line on pg 409 which states, “SIMD hardware can efficiently support SPMD program execution provided that individual threads follow similar control flow.” Isn’t the whole idea of SPMD that various tasks are being run that are very different (i.e. very little similarity in control flow)?

E2.) What exactly is the Needleman-Wunsch algorithm doing, as mentioned on page 411?
NW is used for sequence alignment: it finds a way to align two traces so that there are minimal gaps in the traces. Gaps would be necessary, for example, if trace A executes an insn that never occurs in trace B.

3.) Could you explain the tasks that are used for evaluation? Especially why for the Black and LU tasks, DYNB and PDOM have huge performance gap?

G4.) Section 4.2 "The warp with the older PC still resides in the warp pool, but will no longer be updated..." Does this mean that older warps just sit around indefinitely and accumulate in the warp pool?

R5.) What is the difference between stall and W0 cycles in Figure 16?

R6.) Could you explain in detail about Figure 16? I don't understand why certain warp, like W4 occupies a large portion?

G7.) Section 4.3: "Each cycle, the issue logic searches for or allocates an entry for the PC of each warp entering the scheduler and increments the associated counter with the number of scalar threads joining the warp pool". What is this counter here and what is its purpose?

8.) If one of the branches takes significantly longer time than the other branch, would this grouping technique still be able to reduce branching latency?

9.) In 4.3, could you please explain "In the minority heuristic, warps with the least frequent PCs are given priority with the hope that, by doing so, these warps may eventually catch up and converge with other threads." How do we know the least frequent PC's beforehand?

10.) Could you explain what the need is for swizzling. I don't understand why in their benchmark for bitonic cannot form larger warps?

11.) If there are too many branches, will the warp pool get full and no warp is ready to issue?

E12.) What is a parallel iterative matching allocator? (section 2)

G13.) In 4.1, register file crossbars are mentioned. Could you explain what the purpose of the crossbar is and how the authors fix the ‘drawbacks’ of this with the "lane aware dynamic warp formation"?

14.) Does thread swizzling solve all home lane issues or are there edge cases where it would not help?