GPU architecture:
Revisiting the SIMT execution model

January 2020

Caroline Collange
Inria Rennes – Bretagne Atlantique
https://team.inria.fr/pacap/members/collange/
caroline.collange@inria.fr
Outline

- Running SPMD software on SIMD hardware
  - Context: software and hardware
  - The control flow divergence problem
- Stack-based control flow tracking
  - Stacks
  - Counters
- Path-based control flow tracking
  - The idea: use PCs
  - Implementation: path list
  - Applications
GPU microarchitecture: where it fits

Software

Architecture: **multi-thread** programming model

SIMT microarchitecture

Hardware datapaths: **SIMD** execution units

__global__ void scale(float a, float * X)
{
    unsigned int tid;
    tid = blockIdx.x * blockDim.x + threadIdx.x;
    X[tid] = a * X[tid];
}
Warps

- Threads are grouped into warps of fixed size

Warp 0
T0  T1  T2  T3

Warp 1
T4  T5  T6  T7

An early SIMT architecture. Musée Gallo-Romain de St-Romain-en-Gal, Vienne
Control flow: uniform or divergent

- Control is **uniform** when all threads in the warp follow the same path.

- Control is **divergent** when different threads follow different paths.

```c
x = 0;
// Uniform condition
if(a[x] > 17) {
    x = 1;
}
// Divergent condition
if(tid < 2) {
    x = 2;
}
```

Outcome per thread:

```
Warp: T0  T1  T2  T3
1 1 1 1
1 1 0 0
```

Computer architect view

- SIMD execution inside a warp
  - One SIMD lane per thread
  - All SIMD lanes see the same instruction

- Control-flow differentiation using **execution mask**
  - All instructions controlled with 1 bit per lane
    - 1 → perform instruction
    - 0 → do nothing
Running independent threads in SIMD

- How to keep threads synchronized?
  - Challenge: divergent control flow

Rules of the game
- One thread per SIMD lane
- **Same instruction** on all lanes
- Lanes can be individually disabled with **execution mask**

- Which instruction?
- How to compute execution mask?

```c
x = 0;
// Uniform condition
if(tid > 17) {
    x = 1;
}
// Divergent conditions
if(tid < 2) {
    if(tid == 0) {
        x = 2;
    } else {
        x = 3;
    }
} else {
    x = 3;
}
```
Example: if

Execution mask inside if statement = if condition

```c
x = 0;
   // Uniform condition
   if(a[x] > 17) {
       x = 1;
   }

   // Divergent condition
   if(tid < 2) {
       x = 2;
   }
```

```
if condition: a[x] > 17? T0 T1 T2 T3
           1 1 1 1

Execution mask: 1 1 1 1
```

```
if condition: tid < 2? 1 1 0 0
```

```
Execution mask: 1 1 0 0
```
State of the art in 1804

- The Jacquard loom is a GPU

- Multiple warp threads with per-thread conditional execution
  - Execution mask given by punched cards
  - Supports 600 to 800 parallel threads
Conditionals that no thread executes

- Do not waste energy fetching instructions not executed
  - Skip instructions when execution mask is all-zeroes

```c
x = 0;
// Uniform condition
if(a[x] > 17) {
  x = 1;
} else {
  x = 0;
}
```

```c
// Divergent condition
if(tid < 2) {
  x = 2;
}
```

- Uniform branches are just usual scalar branches
What about loops?

- Keep looping until all threads exit
  - Mask out threads that have exited the loop

```c
i = 0;
while (i < tid) {
    i++;
}
print(i);
```

Execution trace:

- **Warp**
  - T0 T1 T2 T3

- **Time**

No active thread left → restore mask and exit loop

```c
print(i); i=? 0 1 2 3
```
What about nested control flow?

```cpp
x = 0;
// Uniform condition
if(tid > 17) {
    x = 1;
}
// Divergent conditions
if(tid < 2) {
    if(tid == 0) {
        x = 2;
    } else {
        x = 3;
    }
}
```

- We need a generic solution!
Outline

- Running SPMD software on SIMD hardware
  - Context: software and hardware
  - The control flow divergence problem
- Stack-based control flow tracking
  - Stacks
  - Counters
- Path-based control flow tracking
  - The idea: use PCs
  - Implementation: path list
  - Applications
What do *Star Wars* and GPUs have in common?
Answer: Pixar!

- In the early 1980's, the Computer Division of Lucasfilm was designing custom hardware for computer graphics
- Acquired by Steve Jobs in 1986 and became Pixar
- Their core product: the Pixar Image Computer

- This early GPU handles nested divergent control flow!
Pixar Image Computer: architecture overview
The mask stack of the Pixar Image Computer

Code

```c
x = 0;

// Uniform condition
if(tid > 17) {
    x = 1;
}

// Divergent conditions
if(tid < 2) {
    push
    if(tid == 0) {
        push
        x = 2;
        pop
    }
    else {
        push
        x = 3;
        pop
    }
    pop
}
```

Mask Stack

1 activity bit / thread

```
1111
```

```
1111
```

```
1111
```

```
1111
```

Observation: stack content is a histogram

- On structured control flow: columns of 1s
  - A thread active at level $n$ is active at all levels $i<n$
  - Conversely: no “zombie” thread gets revived at level $i>n$ if inactive at $n$
Observation: stack content is a histogram

On structured control flow: columns of 1s
- A thread active at level $n$ is active at all levels $i < n$
- Conversely: no “zombie” thread gets revived at level $i > n$ if inactive at $n$

The height of each column of 1s is enough
- Alternative implementation: maintain an activity counter for each thread

With activity counters

Code

```c
x = 0;

// Uniform condition
if(tid > 17) {
    x = 1;
}

// Divergent conditions
if(tid < 2) {
    inc
    if(tid == 0) {
        inc
        inc
        x = 2;
    dec
    }
    else {
        inc
        x = 3;
    dec
    }
} dec
```

Counters

1 (in)activity counter / thread

<table>
<thead>
<tr>
<th>tid=0</th>
<th>tid=1</th>
<th>tid=2</th>
<th>tid=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0011</td>
<td>0122</td>
<td>0011</td>
</tr>
<tr>
<td>1022</td>
<td>0011</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
Activity counters in use

- In an SIMD prototype from École des Mines de Paris in 1993

- In Intel integrated graphics since 2004
Outline

- Running SPMD software on SIMD hardware
  - Context: software and hardware
  - The control flow divergence problem
- Stack-based control flow tracking
  - Stacks, counters
  - A compiler perspective
- Path-based control flow tracking
  - The idea: use PCs
  - Implementation: path list
  - Applications
- Software approaches
  - Use cases and principle
  - Scalarization
- Research directions
Back to ancient history

Microsoft DirectX

<table>
<thead>
<tr>
<th>7.x</th>
<th>8.0</th>
<th>8.1</th>
<th>9.0a</th>
<th>9.0b</th>
<th>9.0c</th>
<th>10.0</th>
<th>10.1</th>
<th>11</th>
</tr>
</thead>
</table>

Unified shaders

NVIDIA

<table>
<thead>
<tr>
<th>NV10</th>
<th>NV20</th>
<th>NV30</th>
<th>NV40</th>
<th>G70</th>
<th>G80-G90</th>
<th>GT200</th>
<th>GF100</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP 16</td>
<td>Programmable shaders</td>
<td>FP 32</td>
<td>Dynamic control flow</td>
<td>SIMT</td>
<td>CUDA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ATI/AMD

<table>
<thead>
<tr>
<th>R100</th>
<th>R200</th>
<th>R300</th>
<th>R400</th>
<th>R500</th>
<th>R600</th>
<th>R700</th>
<th>Evergreen</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP 24</td>
<td>CTM</td>
<td>FP 64</td>
<td>CAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

GPGPU traction

2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010
Early days of programmable shaders

It is 21\textsuperscript{st} century!

- Graphics cards now look and sound like hair dryers

- Graphics shaders are programmed in assembly-like language
  - Direct3D shader assembly, OpenGL ARB Vertex/Fragment Program...
  - Control-flow: if, else, endif, while, break... are assembly instructions

- Graphics driver performs a straightforward translation to GPU-specific machine language
Goto considered harmful?

Control instructions in some CPU and GPU instruction sets

- GPUs: instruction set expresses control flow structure
  - Where should we stop?
Next: compilers for GPU code

- High-level shader languages
  - C-like: HLSL, GLSL, Cg
  - Then visual languages (UDK)

- General-purpose languages
  - CUDA, OpenCL
  - Then directive-based: OpenACC, OpenMP 4
  - Python (Numba)...

- Incorporate function calls, switch-case, && and ||...

- Demands a compiler infrastructure
  - A Just-In-Time compiler in graphics drivers
A typical GPU compiler

- First: turns all structured control flow into gotos, generates intermediate representation (IR)
  - e.g. Nvidia PTX, Khronos SPIR, llvm IR

- Then: performs various compiler optimizations on IR

- Finally: reconstructs structured control flow back from gotos to emit machine code
  - Not necessarily the same as the original source!
Issues of stack-based implementations

If GPU threads are actual threads, they can synchronize?
- e.g. using semaphores, mutexes, condition variables…
- Problem: SIMT-induced livelock

```c
while(!acquire(lock)) {} 
... 
release(lock)
```

Example: critical section
Thread 0 acquires the lock, keeps looping with other threads of the warp waiting for the lock. Infinite loop, lock never released.

- Stack-based SIMT divergence control can cause starvation!
Issues of stack-based implementations

- Are all control flow optimizations valid in SIMT?

  ```
  f();
  ```

  if(c)
  ```
  f();
  ```

  else
  ```
  f();
  ```

- What about context switches?
  - e.g. migrate one single thread of a warp
  - Challenging to do with a stack

- Truly general-purpose computing demands more flexible techniques
Outline

- Running SPMD software on SIMD hardware
  - Context: software and hardware
  - The control flow divergence problem
- Stack-based control flow tracking
  - Stacks, counters
  - A compiler perspective
- Path-based control flow tracking
  - The idea: use PCs
  - Implementation: path list
  - Applications
With 1 PC / thread

Code

```
x = 0;
if(tid > 17) {
    x = 1;
}
if(tid < 2) {
    if(tid == 0) {
        x = 2;
    } else {
        x = 3;
    }
}
```

Program Counters (PCs)

```
tid= 0 1 2 3
```

- **Match → active**
- **No match → inactive**
Mask stacks vs. per-thread PCs

- Before: stack, counters
  - \(O(n)\), \(O(\log n)\) memory
    \(n = \) nesting depth
  - 1 R/W port to memory
  - Exceptions: stack overflow, underflow

- Vector semantics
  - Structured control flow only
  - Specific instruction sets

- After: multiple PCs
  - \(O(1)\) memory
  - No shared state
  - Allows thread suspension, restart, migration

- Multi-thread semantics
  - Traditional languages, compilers
  - Traditional instruction sets

- Can be mixed with MIMD

- Straightforward implementation is more expensive
Path-based control flow tracking

- A **path** is characterized by a PC and execution mask

\[
\begin{array}{ccccccc}
    & T_0 & T_1 & \ldots & T_7 \\
17 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
\end{array}
\]

- The mask encodes the **set of threads** that have this PC

\{ T_1, T_3, T_4, T_7 \} have PC 17
A list of paths represents a vector of PCs

- Worst case: 1 path per thread
  - Path list size is bounded
- PC vector and path list are equivalent
  - You can switch freely between MIMD thinking and SIMD thinking!
Pipeline overview

- Select an active path

<table>
<thead>
<tr>
<th></th>
<th>$T_0$</th>
<th>$T_1$</th>
<th>$T_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>List of paths</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Pipeline overview

- Select an active path
- Fetch instruction at PC of active path
Pipeline overview

- Select an active path
- Fetch instruction at PC of active path
- Execute with execution mask of active path
Pipeline overview

- Select an active path
- Fetch instruction at PC of active path
- Execute with execution mask of active path
- For uniform instruction: update PC of active path
A divergent branch splits the active path into two paths (or more)
Insert the paths in path list

**List of paths**

<table>
<thead>
<tr>
<th>PC</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>00100010</td>
</tr>
</tbody>
</table>

**Branch not-taken path**

<table>
<thead>
<tr>
<th>PC</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>10000000</td>
</tr>
</tbody>
</table>

**Branch taken path**

<table>
<thead>
<tr>
<th>PC</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>00000100</td>
</tr>
</tbody>
</table>

**Execution mask**

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**New path table**

<table>
<thead>
<tr>
<th>PC</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>00100010</td>
</tr>
<tr>
<td>12</td>
<td>10000000</td>
</tr>
<tr>
<td>17</td>
<td>01011001</td>
</tr>
<tr>
<td>17</td>
<td>00000100</td>
</tr>
</tbody>
</table>
Convergence is path fusion

- When two paths have the same PC, we can merge them
  - New set of threads is the union of former sets
  - New execution mask is bitwise OR of former masks

\[
\begin{array}{cccccccc}
4 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
12 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
17 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
17 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\quad \rightarrow \quad
\begin{array}{cccccccc}
4 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
12 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
17 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
17 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
\end{array}
\]
Path scheduling is graph traversal

- Degrees of freedom
  - Which path is the active path?
  - At which place are new paths inserted?
  - When and where do we check for convergence?

- Different answers yield different policies
Depth-first graph traversal

- Remember graph algorithm theory
  - Depth-first graph traversal using a stack worklist
- Path list as a stack
  = depth-first traversal of the control-flow graph
  - Most deeply nested levels first

```python
x = 0;
// Uniform condition
if(tid > 17) {
    x = 1;
}
// Divergent conditions
if(tid < 2) {
    if(tid == 0) {
        x = 2;
    } else {
        x = 3;
    }
} else {
    x = 3;
}
```

Question: is this the same as Pixar-style mask stack? Why?
Breadth-first graph traversal

- Goal: guarantee forward progress to avoid SIMT-induced livelocks

```c
while (!acquire(lock)) {
    1:
    }
2: ...
    release(lock)
3:
```

- Path list as a queue: follow paths in round-robin

- Drawback: may delay convergence

Example: Nvidia Volta (2017)

Supports independent thread scheduling inside a warp

- Threads can synchronize with each other inside a warp
- Diverged threads can run barriers (as long as all threads eventually reach a barrier)
Advertisement: Simty, a SIMT CPU

- Proof of concept for priority-based SIMT
  - Written in synthesizable VHDL
  - Runs the RISC-V instruction set (RV32I)
  - Fully parametrizable warp size, warp count
  - 10-stage pipeline

https://team.inria.fr/pacap/simty/
Programmer's view

- Programming model
  - SPMD: Single program, multiple data
  - One \textit{kernel} code, many \textit{threads}
  - Unspecified execution order between explicit synchronization barriers

- Languages
  - Graphics shaders: HLSL, Cg, GLSL
  - GPGPU: C for CUDA, OpenCL

\begin{equation}
\text{For } n \text{ threads:} \\
X[tid] \leftarrow a \times X[tid]
\end{equation}

Kernel
Types of control flow

- **Structured** control flow:
  - single-entry, single exit properly nested
    - Conditionals: `if-then-else`
    - Single-entry, single-exit loops: `while, do-while, for...`
    - Function call-return...

- **Unstructured** control flow
  - `break, continue`
  - `&& ||` short-circuit evaluation
  - Exceptions
  - Coroutines
  - `goto, comefrom`
  - Code that is hard to indent!