Architecture and micro-architecture of GPUs

Sylvain Collange
Arénaire, LIP, ENS de Lyon
sylvain.collange@ens-lyon.fr

Departamento de Ciência da Computação,
UFMG - ICEX
May 27, 2011
Where I come from

- Arénaire, LIP, École normale supérieure de Lyon
  11 faculty members, 17 PhD/postdoc/staff
  - Computer arithmetic, at all levels
    - Validated algorithms
    - Function approximation in software
    - Hardware operators on FPGAs…
  - http://www.ens-lyon.fr/LIP/Arenaire/

- DALI, LIRMM, Université de Perpignan
  7 faculty members, 6 PhD/postdoc
  - Computer arithmetic
    - Compensation, certification
  - Computer architecture
    - Simulation, micro-architecture, GPUs
  - http://webdali.univ-perp.fr/
From GPU to integrated many-core

- Yesterday (2000-2010)
  - Homogeneous multi-core
  - Discrete components

- Today (2011-...)
  Heterogeneous multi-core
  - Intel Sandy Bridge
  - AMD Fusion
  - NVIDIA Denver/Maxwell project...

- Focus on the throughput-optimized part
  - Similarities?
  - Differences?
  - Possible improvements?
Outline

- Performance or efficiency?
  - Latency architecture
  - Throughput architecture
- Execution units: efficiency through regularity
  - Traditional divergence control
  - Towards more flexibility
- Memory access: locality and regularity
  - Some memory organizations
  - Maximizing throughput
The 1980's: pipelined processor

- Example: scalar-vector multiplication: \( X \leftarrow a \cdot X \)

for \( i = 0 \) to \( n-1 \)
\[
X[i] \leftarrow a \ast X[i]
\]

Source code

move \( i \leftarrow 0 \)

loop:
load \( t \leftarrow X[i] \)
mul \( t \leftarrow a \times t \)
store \( X[i] \leftarrow t \)
add \( i \leftarrow i+1 \)
branch \( i < n ? \) loop

Machine code

Sequential CPU

Memory

- Fetch
- Decode
- Execute
- L/S Unit
The 1990's: superscalar processor

- **Goal:** improve performance of sequential applications
  - Latency: time to get the result
- **Exploits Instruction-Level Parallelism (ILP)**
- **Lots of tricks**
  - Branch prediction, out-of-order execution, register renaming, data prefetching, memory disambiguation…
- **Basis: speculation**
  - Take a bet on future events
  - If right: time gain
  - If wrong, roll back: energy loss
What makes speculation work: regularity

- Application behavior likely to follow regular patterns

Control regularity

```c
for(i...)
{
  if(f(i)) {
  }
}
```

Memory regularity

```c
j = g(i);
x = a[j];
```

Regular case

<table>
<thead>
<tr>
<th>Time</th>
<th>i=0</th>
<th>i=1</th>
<th>i=2</th>
<th>i=3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
</tbody>
</table>

Irregular case

<table>
<thead>
<tr>
<th></th>
<th>i=0</th>
<th>i=1</th>
<th>i=2</th>
<th>i=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>not tk</td>
<td>taken</td>
<td>taken</td>
<td>not tk</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>j=17</th>
<th>j=18</th>
<th>j=19</th>
<th>j=20</th>
</tr>
</thead>
<tbody>
<tr>
<td>j=21</td>
<td>j=4</td>
<td>j=17</td>
<td>j=2</td>
</tr>
</tbody>
</table>

Applications

- Caches
- Branch prediction
- Instruction prefetch, data prefetch, write combining…
The 2000's: going multi-threaded

- Obstacles to continuous CPU performance increase
  - Power wall
  - Memory wall
  - ILP wall
- 2000-2010: gradual transition from latency-oriented to throughput-oriented
  - Homogeneous multi-core
  - Interleaved multi-threading
  - Clustered multi-threading
Homogeneous multi-core

- **Replication** of the complete execution engine
- Multi-threaded software

```
move  i ← slice_begin
loop:
  load  t ← X[i]
  mul   t ← a×t
  store X[i] ← t
  add   i ← i+1
branch i<slice_end? loop
```

**Machine code**

Threads:  T0  T1

Improves throughput thanks to explicit parallelism
Interleaved multi-threading

- **Time-multiplexing** of processing units
- **Same software view**

```move
move i ← slice_begin

loop:
load t ← X[i]
mul t ← a×t
store X[i] ← t
add i ← i+1
branch i<slice_end? loop
```

Machine code

- Hides latency thanks to explicit parallelism
Clustered multi-core

- For each individual unit, select between
  - Replication
  - Time-multiplexing

- Examples
  - Sun UltraSparc T2
  - AMD Bulldozer

- Area-efficient tradeoff
Outline

- Performance or efficiency?
  - Latency architecture
  - Throughput architecture
- Execution units: efficiency through regularity
  - Traditional divergence control
  - Towards more flexibility
- Memory access: locality and regularity
  - Some memory organizations
  - Maximizing throughput
Why heterogeneity?

- Amdahl's law
  \[ S = \frac{1}{(1-P) + \frac{P}{N}} \]
  - Time to run sequential portions
  - Time to run parallel portions

- Latency-optimized multi-core
  - Low efficiency on parallel portions: spends too much resources

- Throughput-optimized multi-core
  - Low performance on sequential portions

- Heterogeneous multi-core
  - Suggests more radical specialization
  - Power-constrained: can afford idle transistors
Threading granularity

- Coarse-grained threading
  - **Decouple** tasks to reduce **conflicts** and inter-thread communication

- Fine-grained threading
  - **Interleave** tasks
  - Exhibit **locality**: neighbor threads share memory
  - Exhibit **regularity**: neighbor threads have a similar behavior
## Parallel regularity

- Similarity in behavior between threads

<table>
<thead>
<tr>
<th></th>
<th>Regular</th>
<th>Irregular</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Control regularity</strong></td>
<td>![Diagram of regular control flow]</td>
<td>![Diagram of irregular control flow]</td>
</tr>
<tr>
<td><strong>Thread</strong></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><strong>i</strong></td>
<td>i=17</td>
<td>i=17</td>
</tr>
<tr>
<td><strong>switch(i) {</strong></td>
<td>![Diagram of control statements]</td>
<td>![Diagram of control statements]</td>
</tr>
<tr>
<td><strong>case 2:</strong></td>
<td>![Diagram of control statements]</td>
<td></td>
</tr>
<tr>
<td><strong>case 17:</strong></td>
<td>![Diagram of control statements]</td>
<td></td>
</tr>
<tr>
<td><strong>case 21:</strong></td>
<td>![Diagram of control statements]</td>
<td></td>
</tr>
<tr>
<td><strong>}</strong></td>
<td>![Diagram of control statements]</td>
<td></td>
</tr>
<tr>
<td><strong>Memory regularity</strong></td>
<td>![Diagram of memory access]</td>
<td>![Diagram of memory access]</td>
</tr>
<tr>
<td><strong>A</strong></td>
<td>![Diagram of memory access]</td>
<td></td>
</tr>
<tr>
<td><strong>load A[i]</strong></td>
<td>![Diagram of memory access]</td>
<td></td>
</tr>
<tr>
<td><strong>r=A[i]</strong></td>
<td>![Diagram of memory access]</td>
<td></td>
</tr>
<tr>
<td><strong>data regularity</strong></td>
<td>![Diagram of data access]</td>
<td>![Diagram of data access]</td>
</tr>
<tr>
<td><strong>a</strong></td>
<td>a=32</td>
<td>a=17</td>
</tr>
<tr>
<td><strong>b</strong></td>
<td>b=52</td>
<td>b=15</td>
</tr>
<tr>
<td><strong>r=a*b</strong></td>
<td>![Diagram of data access]</td>
<td></td>
</tr>
<tr>
<td><strong>A</strong></td>
<td>![Diagram of memory access]</td>
<td></td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>![Diagram of memory access]</td>
<td></td>
</tr>
</tbody>
</table>
Single Instruction, Multiple Threads (SIMT)

- **Cooperative sharing** of fetch/decode, load-store units
  - Fetch 1 instruction on behalf of several threads
  - Read 1 memory location and broadcast to several registers

In NVIDIA-speak
- SIMT: Single Instruction, Multiple Threads
- Convoy of synchronized threads: *warp*

- Improves Area/Power-efficiency thanks to **regularity**
  - Consolidates memory transactions: less memory pressure
Example GPU: NVIDIA GeForce GTX 580

- SIMT: warps of 32 threads
- 16 SMs / chip
- 2×16 cores / SM, 48 warps / SM

- 1580 Gflop/s
- Up to 24576 threads in flight
Outline

- Performance or efficiency?
  - Latency architecture
  - Throughput architecture
- Execution units: efficiency through regularity
  - Traditional divergence control
  - Towards more flexibility
- Memory access: locality and regularity
  - Some memory organizations
  - Maximizing throughput
Capturing instruction regularity

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;
}

- How to handle control divergence?
- Rules of the game
  - One thread per Processing Element (PE)
  - All PE execute the same instruction
  - PEs can be individually disabled

Techniques from Single Instruction, Multiple Data architectures
Most common: mask stack

Code

```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;
}
```

Mask Stack

1 activity bit / thread

```
1111
```

tid=0  tid=2

tid=1  tid=3

Activity counters

Code

```c
x = 0;

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

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

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></td>
</tr>
</tbody>
</table>

1 Program Counter / thread

Code

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

Program Counters (PCs)

<table>
<thead>
<tr>
<th>tid=</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
</table>

Match → active

No match → inactive

Master PC

PC0

PC1

PC2

PC3

How to compute the Master PC?

- Early SIMD machines (1980 – 1985)
  - Software or hardware
  - Master runs through all conditional blocks, whether taken or not
    Runs through loop bodies $n+1$ times

  - Mostly software (with hardware support)
  - Only runs through branches actually taken

- Current GPUs (2005 – 2010)
  - In hardware
  - Structured control-flow in the instruction set
Traditional SIMT pipeline

- Used in virtually all current GPUs
Outline

- Performance or efficiency?
  - Latency architecture
  - Throughput architecture

- Execution units: efficiency through regularity
  - Traditional divergence control
  - Towards more flexibility

- Memory access: locality and regularity
  - Some memory organizations
  - Maximizing throughput
Goto considered harmful?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>bar</td>
<td>jmpi</td>
<td>jmpi</td>
<td>jump</td>
<td>push</td>
<td>push</td>
<td>push</td>
</tr>
<tr>
<td>jal</td>
<td>bra</td>
<td>if</td>
<td>if</td>
<td>loop</td>
<td>push_else</td>
<td>push_else</td>
<td>push else</td>
</tr>
<tr>
<td>jr</td>
<td>brk</td>
<td>else</td>
<td>else</td>
<td>endloop</td>
<td>pop</td>
<td>pop</td>
<td>pop</td>
</tr>
<tr>
<td>syscall</td>
<td>brkpt</td>
<td>endif</td>
<td>endif</td>
<td>rep</td>
<td>loop_start</td>
<td>loop_start_no_al</td>
<td></td>
</tr>
<tr>
<td>cal</td>
<td>cont</td>
<td>do</td>
<td>while</td>
<td>endrep</td>
<td>loop_start_dx10</td>
<td>loop_start_dx10</td>
<td></td>
</tr>
<tr>
<td>kil</td>
<td>exit</td>
<td>break</td>
<td>break</td>
<td>breakloop</td>
<td>loop_end</td>
<td>loop_end</td>
<td></td>
</tr>
<tr>
<td>pbk</td>
<td>jcal</td>
<td>cont</td>
<td>cont</td>
<td>breakloop</td>
<td>loop_continue</td>
<td>loop_continue</td>
<td></td>
</tr>
<tr>
<td>pret</td>
<td>jmx</td>
<td>halt</td>
<td>halt</td>
<td>breakrep</td>
<td>loop_break</td>
<td>loop_break</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>kil</td>
<td>msave</td>
<td>msave</td>
<td>return</td>
<td>jump</td>
<td>jump</td>
<td></td>
</tr>
<tr>
<td>ssy</td>
<td>pbk</td>
<td>mrest</td>
<td>mrest</td>
<td>return</td>
<td>else</td>
<td>else</td>
<td></td>
</tr>
<tr>
<td>trap</td>
<td>pret</td>
<td>push</td>
<td>push</td>
<td>return_fs</td>
<td>call</td>
<td>call</td>
<td></td>
</tr>
<tr>
<td>.s</td>
<td>ret</td>
<td>pop</td>
<td>pop</td>
<td>return_fs</td>
<td>call_fs</td>
<td>call_fs</td>
<td></td>
</tr>
<tr>
<td>.s</td>
<td>ssy</td>
<td></td>
<td></td>
<td></td>
<td>return</td>
<td>return</td>
<td></td>
</tr>
<tr>
<td>.s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>return_fs</td>
<td>return_fs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>alu</td>
<td>alu</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>alu_push_before</td>
<td>alu_push_before</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>alu_pop_after</td>
<td>alu_pop_after</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>alu_pop2_after</td>
<td>alu_pop2_after</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>alu_continue</td>
<td>alu_continue</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>alu_break</td>
<td>alu_break</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>alu_else_after</td>
<td>alu_else_after</td>
<td></td>
</tr>
</tbody>
</table>

Control instructions in some CPU and GPU instruction sets
SIMD is so last century

- **Maspar MP-1 (1990)**
  - 1 instruction for 16,384 PEs
  - PE: ~1 mm², 1.6 μm process
  - SIMD programming model

- **NVIDIA Fermi (2010)**
  - 1 instruction for 16 PEs
  - PE: ~0.03 mm², 40 nm process
  - Threaded programming model

→ From centralized control to flexible distributed control
A democratic instruction sequencer

- Maintain one PC per thread
- Vote: select one of the individual PCs as the Master PC
- Which one? Various policies:
  - Majority: most common PC
  - Minimum: thread that is late
  - Deepest control flow nesting level
  - Deepest function call nesting level
  - Various combinations of the former

S. Collange. Une architecture unifiée pour traiter la divergence de contrôle et la divergence mémoire en SIMT. SympA'14, 2011.
Our new SIMT pipeline

- **Instruction Fetch**
  - $PC_0$ → Vote → Instruction Fetch
  - $PC_1$ → Vote → Instruction Fetch
  - $PC_n$ → Vote → Instruction Fetch

- **Broadcast**
  - Match → Exec → Update PC → $PC_0$
  - Match → Exec → Update PC → $PC_1$
  - Match → Exec → Update PC → $PC_n$

- **No match: discard instruction**
Benefits of multiple-PC arbitration

- Before: stack, counters
  - $O(n)$, $O(\log n)$ memory
    $n = \text{nesting depth}$
  - 1 R/W port to memory
  - Exceptions: stack overflow, underflow
- Still SIMD semantics (Bougé-Levaire)
  - Structured control flow only
  - Specific instruction sets

- After: multiple PCs
  - $O(1)$ memory
  - No shared state
  - Allows thread suspension, restart, migration
- True SPMD semantics (multi-thread)
  - Traditional languages, compilers
  - Traditional instruction sets
- Enables many new architecture ideas
With multiple warps

- Two-stage scheduling
  - Select one warp
  - Select one instruction (MPC) for this warp

<table>
<thead>
<tr>
<th>Warp</th>
<th>Ready?</th>
<th>Next I</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>✓</td>
<td>sub r4, r0</td>
</tr>
<tr>
<td>1</td>
<td>✓</td>
<td>add r1, r3</td>
</tr>
<tr>
<td>2</td>
<td>✗</td>
<td>load r3, [r1]</td>
</tr>
<tr>
<td>3</td>
<td>✗</td>
<td>mul r5, r2</td>
</tr>
</tbody>
</table>

Lane:

- 0: mul, add, sub, mul
- 1: add, mul, add, add
- 2: mul, add, store, load
- 3: load, mul, mul, add

Read registers

Execute
Dual Instruction, Multiple Threads (DIMT)

- Two-stage scheduling
  - Select one warp
  - Select **two** instructions ($\text{MPC}_1$, $\text{MPC}_2$) for this warp

Warp | Ready? | Next I
--- | --- | ---
0 | ✓ | sub r4, r0 | add r1, r3 | mul | add | sub | mul
1 | ✓ | add r1, r3 | mul r5, r2 | add | mul | add | add
2 | x | load r3, [r1] | add r1, r3 | mul | add | store | load
3 | x | mul r5, r2 | add r1, r3 | load | mul | mul | add

- More than 2 instructions: NIMT

Why DIMT?

- “Fills holes” using parallelism between execution paths.

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

<table>
<thead>
<tr>
<th>Program Counters (PCs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tid= 0 1 2 3</td>
</tr>
</tbody>
</table>

- No overlap

Master PC 0}

Master PC 1

PC0

PC1

PC2 PC3
Dynamic Warp Formation (DWF)

- Why need warps at all?
  - Select master PC from global thread pool
  - On each PE, select one thread from local thread pool

![Diagram showing Dynamic Warp Formation](image)

New DIMT+DWF pipeline

- Radical departure from classical SIMD
Avoiding redundancy

- Goal: maximize execution unit utilization?

- Maximize execution unit utilization doing real work!
What are we computing on?

- **Uniform data**
  - In a warp, $v[i] = c$

- **Affine data**
  - In a warp, $v[i] = b + i \cdot s$
  - Base $b$, stride $s$

- **Average frequency on GPGPU applications**
Tagging registers

- Associate a tag to each vector register
  - Uniform, Affine, unKnown
- Propagate tags across arithmetic instructions
- 2 lanes are enough to encode uniform and affine vectors

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov i ← tid</td>
<td>A←A</td>
</tr>
<tr>
<td>loop:</td>
<td></td>
</tr>
<tr>
<td>load t ← X[i]</td>
<td>K←U[A]</td>
</tr>
<tr>
<td>mul t ← a×t</td>
<td>K←U×K</td>
</tr>
<tr>
<td>store X[i] ← t</td>
<td>U[A]←K</td>
</tr>
<tr>
<td>add i ← i+tcnt</td>
<td>A←A+U</td>
</tr>
<tr>
<td>branch i&lt;n? loop</td>
<td>A&lt;U?</td>
</tr>
<tr>
<td>loop:</td>
<td></td>
</tr>
<tr>
<td>load t ← X[i]</td>
<td>K←U[A]</td>
</tr>
<tr>
<td>mul t ← a×t</td>
<td>K←U×K</td>
</tr>
</tbody>
</table>

...
Dynamic Work Factorization (DWF)

Inactive for 38% of operands

Inactive for 24% of instructions

S. Collange, D. Defour, Y. Zhang. *Dynamic detection of uniform and affine vectors in GPGPU computations*. Europar HPPC09, 2009
Control logic needs to stay much smaller / simpler / less power-hungry than Execution logic.

Is execution unit utilization such an issue anyway?
Outline

- Performance or efficiency?
  - Latency architecture
  - Throughput architecture
- Execution units: efficiency through regularity
  - Traditional divergence control
  - Towards more flexibility
- Memory access: locality and regularity
  - Some memory organizations
  - Maximizing throughput
It is all about memory

- Our primary constraint: power
- Power measurements on NVIDIA GT200

<table>
<thead>
<tr>
<th></th>
<th>Energy/op (nJ)</th>
<th>Total power (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction control</td>
<td>1.8</td>
<td>18</td>
</tr>
<tr>
<td>Multiply-add on a 32-wide warp</td>
<td>3.6</td>
<td>36</td>
</tr>
<tr>
<td>Load 128B from DRAM</td>
<td>80</td>
<td>90</td>
</tr>
</tbody>
</table>

- With the same amount of energy
  - Load 1 word from DRAM
  - Compute 44 flops

- Memory traffic is what matters

Memory access patterns

In traditional vector processing

Scalar load & broadcast
Reduction & scalar store

(Non-unit) strided load
(Non-unit) strided store

Unit-strided load
Unit-strided store

In SIMT

- Every load is a gather, every store is a scatter
The memory we want

- Many independent R/W ports
- Supports lots of small transactions: 4B or 8B-wide
The memory we have

- DRAMs
  - Wide bus, burst mode
    - Use **wide transactions** (≥32B)
  - Switching pages is expensive
    - **Group** accesses by pages (1 page ~ 2KB)
  - **One shared bus**, read/write turnaround penalty
    - **Group** accesses by direction

- Caches
  - Have wide cache lines (128B-256B)
  - Have **few R/W ports**
Breakdown of memory access patterns

- Vast majority: uniform or unit-strided
  - And even aligned vectors

“In making a design trade-off, favor the frequent case over the infrequent case.” [HP06]
Coalescing concurrent requests

- **Unit-strided detection (NVIDIA CC 1.0-1.1 coalescing)**

  1. Select one request, consider maximal aligned transaction
  2. Identify requests that fall in the same memory segment
  3. Reduce transaction size when possible and issue transaction
  4. Repeat with remaining requests

  → One transaction

  → Multiple transactions

- **Minimal coverage (NVIDIA CC 1.2 coalescing)**

  1. Select one request, consider maximal aligned transaction
  2. Identify requests that fall in the same memory segment
  3. Reduce transaction size when possible and issue transaction
  4. Repeat with remaining requests
Banked shared memory

- Software-managed memory
- Interleaved on a word-by-word basis

Used in NVIDIA Tesla (2007)
Hardware-managed cache

- Share one wide port to the L1 cache
- Multiple lanes can read from the same cache line
- Bottleneck: single-ported cache tags

Used in NVIDIA Fermi (2010)
Outline

- Performance or efficiency?
  - Latency architecture
  - Throughput architecture
- Execution units: efficiency through regularity
  - Traditional divergence control
  - Towards more flexibility
- Memory access: locality and regularity
  - Some memory organizations
  - Maximizing throughput
Dealing with pipeline hazards

- Bank conflicts
- Lost arbitration
- Cache misses

- Conventional solution: stall execution pipeline until resolved
Preferred solution: in-order replay

- Instruction replay
  - Keep pipeline running
  - Put back offending instruction in instruction queue
  - With updated pred mask: only replay threads that failed

Used in NVIDIA Tesla (2007)
Dynamic Warp Subdivision

- Consider Replay as a control-flow operation (or no-op)
  - Threads that miss are turned inactive until data arrives
  - Threads that hit ask for next instruction
- Memory divergence = branch divergence
  - Both handled the same way
- When one thread misses, no need to block the whole warp
- Tradeoff: more latency hiding, lower ALU utilization
  - Can counteract utilization loss with DIMT/NIMT

Linked list traversal: without DWS

1: while(i != -1) {
2:   i = l[i];
3: }

Thread 0:

PC=2
2: i = l[i];
hit

MPC=2

Thread 1:

PC=2
2: i = l[i];
miss

Thread 2:

PC=2
2: i = l[i];
hit

Thread 3:

PC=2
hit

---

MPC=1

1: i != -1?
true

MPC=2

2: i = l[i];
miss

MPC=1

1: i != -1?
true

MPC=1

1: i != -1?
false
Linked list traversal: with DWS

1: while(i != -1) {
2:   i = l[i];
3: }

MPC=2
2: i = l[i];

MPC=1
1: i != -1?

MPC=2
2: i = l[i];

MPC=1
1: i != -1?

MPC=2
2: i = l[i];

MPC=1
1: i != -1?
SIMT pipeline – memory instruction

Hazards: Divergence  Bank conflict  Cache miss
all cause PC and valid bit to be updated accordingly
Conclusion and future challenges

- SIMT bridges the gap between superscalar and SIMD
  - Smooth, dynamic tradeoff between regularity and efficiency
- Future micro-architectural improvements?

![Graph showing efficiency (flops/W) vs regularity of application. The graph compares superscalar, SIMT, and SIMD architectures. Notable improvements include dynamic warp formation, dynamic warp subdivision, and NIMT.](image)
Architecture and micro-architecture of GPUs

Sylvain Collange
Arénaire, LIP, ENS de Lyon
sylvain.collange@ens-lyon.fr

Departamento de Ciência da Computação,
UFMG - ICEx
May 27, 2011
Software view

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

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

![Diagram showing synchronization barriers and code execution]

For n threads:

```
X[tid] ← a * X[tid]
```
void scale(float a, float * X, int n) {
    for(int i = 0; i != n; ++i)
        X[i] = a * X[i];
}
__global__ void scale(float a, float * X) {
    unsigned int tid;
    tid = blockIdx.x * blockDim.x + threadIdx.x;
    X[tid] = a * X[tid];
}

Architecture: multi-thread programming model

SIMT microarchitecture

Hardware datapaths: SIMD execution units

GPU microarchitecture
The old way: centralized control

- Scalar unit takes all control decisions
  - Gives instructions
- SIMD / Vector units act as a coprocessor
  - Reports hazards
The new way: piggyback model

- Multiple threads, one shared resource
  - Instruction fetch, memory bank port, cache tag port...
- Select one thread, give it access to the resource
- Let other threads opportunistically share the resource
  - Same instruction, same cache line...
- Variations: multiple resources, grouping strategies, arbitration policies...
Sequential case: cache and prefetch

Code
for i = 0 to N-1
load X[i]

Trace
load X[i] Cache miss
load X[i]
load X[i]
load X[i]
load X[i]

Memory
Fetch cache line
Prefetch next line

• “Vertical” temporal or spatial locality
  ♦ Sequential reuse, in time
Parallel case: coalescing, multi-threading

**Code**

```
kernel:
  load X[tid]
  ...
```

**Trace**

- **w0**: load $X\{0,1,2,3\}$
- **w1**: load $X\{4,5,6,7\}$
- **w2**: load $X\{8,9,10,11\}$

- “Horizontal” temporal or spatial locality
  - Parallel reuse, in space
On-chip memory

- Conventional wisdom
  - Cache area in CPU vs. GPU according to the NVIDIA CUDA Programming Guide:

- Actual data

<table>
<thead>
<tr>
<th>GPU</th>
<th>Register files + caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA GF110</td>
<td>3.9 MB</td>
</tr>
<tr>
<td>AMD Cayman</td>
<td>7.7 MB</td>
</tr>
</tbody>
</table>

- At this rate, will catch up with CPUs by 2012…
The cost of SIMT: register wastage

SIMD

mov i ← 0
loop:
  vload T ← X[i]
  vmul T ← a×T
  vstore X[i] ← T
  add i ← i+16
  branch i<n? loop

SIMT

mov i ← tid
loop:
  load t ← X[i]
  mul t ← a×t
  store X[i] ← t
  add i ← i+tcnt
  branch i<n? loop

- Instructions

  vload  T ← X[i]
  vmul  T ← a×T
  vstore X[i] ← T
  add   i ← i+16
  branch i<n? loop

  load   t ← X[i]
  mul    t ← a×t
  store  X[i] ← t
  add    i ← i+tcnt
  branch i<n? loop

- Registers

  T
  a
  i
  n
  scalar
  vector

  t
  a
  i
  n
  scalar
  vector
So how does it differ from SIMD?

<table>
<thead>
<tr>
<th></th>
<th>SIMD</th>
<th>SIMT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction regularity</strong></td>
<td>Vectorization at compile-time</td>
<td>Vectorization at runtime</td>
</tr>
<tr>
<td><strong>Control regularity</strong></td>
<td>Software-managed Bit-masking, predication</td>
<td>Hardware-managed Stack, counters, multiple PCs</td>
</tr>
<tr>
<td><strong>Memory regularity</strong></td>
<td>Compiler selects: vector load-store or gather-scatter</td>
<td>Hardware-managed Gather-scatter with hardware coalescing</td>
</tr>
</tbody>
</table>

- SIMD is static, SIMT is dynamic
- Similar opposition as VLIW vs. superscalar
Dynamic data deduplication: results

- Detects
  - 79% of affine input operands
  - 75% of affine computations

![Bar chart showing the percentage of inputs and outputs detected and total inputs and outputs, with categories of Unknown, Affine, and Uniform.]
My register file is full of holes!

- Static partitioning of RF between warps is space-inefficient

  ![Diagram showing register allocation across warps](image)

- We need more dynamic, flexible register allocation
  - AMD: more flexible, but static
  - Need to take into account affine registers

- Solution: play Tetris?
  - Or use caches
Ongoing work: affine caches

- As level-1 cache
  - Extension to the RF: 1 cache line = 1 spilled vector register
  - Space-efficient storage of call stacks

- Inspired from zero-content augmented caches [DPS09]
Affine cache as L0: RF replacement

- SIMT execution: only 1 tag lookup / operand
  - Same translation for all lanes
- Affine ALU handles most control flow and addresses
- Vector ALUs/FPUs do the heavy lifting
- Coordinate warp scheduling and cache replacement?
Focus on GPGPU

- Graphics Processing Unit (GPU)
  - Video game industry: mass market
  - Low unit price, amortized R&D
    - Inexpensive, high-performance parallel processor
- 2002: General-Purpose computation on GPU (GPGPU)
- 2010: #1 supercomputer
  - Tianhe-1A supercomputer
  - 7168 GPUs (NVIDIA Tesla M2050)
  - 2.57 Pflops
  - 4.04 MW “only”
  - #1 in Top500, #11 in Green500

Credits: NVIDIA
**SIMD**

- Single Instruction Multiple Data

```plaintext
for i = 0 to n-1 step 4
    X[i..i+3] ← a * X[i..i+3]
```

Source code

```plaintext
loop:
    vload  T ← X[i]
    vmul   T ← a*T
    vstore X[i] ← T
    add    i ← i+4
    branch i<n? loop
```

Machine code

**Challenging to program (semi-regular apps?)**