Escaping the SIMD vs. MIMD mindset
A new class of hybrid microarchitectures between GPUs and CPUs

Sylvain Collange
Università degli Studi di Siena
sylvain.collange@gmail.com

Séminaire DALI
December 15, 2011
This talk is *not* about GPUs

- Yesterday (2000-2010)
  - Homogeneous multi-core
  - Discrete components
- Today (2011-...)
  Chip-level integration
  - Intel Sandy Bridge
  - AMD Fusion
  - NVIDIA Denver/Maxwell project…
- Tomorrow
  Heterogeneous multi-core
  - Focus on the throughput-optimized part
  - Programming model: SPMD
Outline

- SIMT architectures
  - Parallel locality and its exploitation
  - Revisiting Flynn's Taxonomy
- How to keep threads synchronized
- Two instructions, multiple data
- Parallel value locality
Locality, regularity in sequential apps

- Application behavior likely to follow regular patterns

```
for(i...) {
    if(f(i)) {
        j = g(i);
        x = a[j];
    }
}
```

### Control regularity

<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>Regular</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>Irregular</td>
<td>not tk</td>
<td>taken</td>
<td>taken</td>
<td>not tk</td>
</tr>
</tbody>
</table>

### Memory locality, regularity

<table>
<thead>
<tr>
<th>Value</th>
<th>i=0</th>
<th>i=1</th>
<th>i=2</th>
<th>i=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>x=15</td>
<td>x=0</td>
<td>x=2</td>
<td>x=52</td>
</tr>
<tr>
<td>Irregular</td>
<td>x=42</td>
<td>x=42</td>
<td>x=42</td>
<td>x=42</td>
</tr>
</tbody>
</table>

### Applications

- Caches
- Branch prediction
- Instruction prefetch, data prefetch, write combining…
Regularity in parallel applications

- Similarity in behavior between SPMD threads

<table>
<thead>
<tr>
<th>Parallel control regularity</th>
<th>Regular</th>
<th>Irregular</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$i=17$</td>
<td>$i=17$</td>
<td>$i=17$</td>
</tr>
<tr>
<td>Switch(i)</td>
<td>{</td>
<td>\begin{align*} &amp;\text{case 2:...} \ &amp;\text{case 17:...} \ &amp;\text{case 21:...} \end{align*}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parallel memory locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \quad \text{Memory} )</td>
</tr>
<tr>
<td>\begin{itemize} \item \text{load} \ A[8] \item \text{load} \ A[9] \item \text{load} \ A[10] \item \text{load} \ A[11] \end{itemize}</td>
</tr>
<tr>
<td>( r=A[i] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parallel value locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>\begin{itemize} \item \text{a}=32 \item \text{a}=32 \item \text{a}=32 \item \text{a}=32 \end{itemize}</td>
</tr>
<tr>
<td>\begin{itemize} \item \text{b}=52 \item \text{b}=52 \item \text{b}=52 \item \text{b}=52 \end{itemize}</td>
</tr>
<tr>
<td>( r=a*b )</td>
</tr>
</tbody>
</table>
How to exploit parallel locality?

Multi-threading implementation options:

- Replication
  - Different resources, same time
  - Chip Multi-Processing (CMP)

- Time-multiplexing
  - Same resource, different times
  - Multi-Threading (MT)

- Factorization
  - If we have parallel locality
  - Same resource, same time
  - Single-Instruction Multi-Threading (SIMT)
Single Instruction, Multiple Threads (SIMT)

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

- Area/Power-efficient thanks to parallel locality
Flynn's taxonomy revisited

<table>
<thead>
<tr>
<th>Resource: pipeline stage</th>
<th>Instruction</th>
<th>RF, Execute</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single resource</td>
<td>Fetch</td>
<td>(Data)</td>
<td>(Address)</td>
</tr>
<tr>
<td>T0 T1 T1 T2 T3 SIMT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple resources</td>
<td>Fetch</td>
<td>(Data)</td>
<td>(Address)</td>
</tr>
<tr>
<td>T0 T1 T1 T2 T3 SIMT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T0 T1 T1 T2 T3 SDMT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T0 T1 T1 T2 T3 SAMT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Mostly orthogonal
  - Mix and match to build your own _I_D_A_T pipeline!
Examples: conventional design points

- Multi-core
  - MIMD(MAMT)

- Short-vector SIMD
  - SIMD(SAST)

- GPU
  - SI(MDSA)MT
A 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

- SIMT architectures
- How to keep threads synchronized
  - The old way: mask stacks
  - The new way: distributed control and arbitration
- Two instructions, multiple data
- Parallel value locality
How to keep threads synchronized?

- **Issue: control divergence**

- **Rules of the game**
  - One thread per Processing Element (PE)
  - All PE execute the same instruction
  - PEs can be individually disabled

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

Code
\[ x = 0; \]
// 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

<table>
<thead>
<tr>
<th>tid=0</th>
<th>tid=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>1100</td>
</tr>
<tr>
<td>1111</td>
<td>1100</td>
</tr>
<tr>
<td>1000</td>
<td>1111</td>
</tr>
<tr>
<td>1100</td>
<td>1111</td>
</tr>
<tr>
<td>0100</td>
<td>1111</td>
</tr>
<tr>
<td>1111</td>
<td>1111</td>
</tr>
</tbody>
</table>
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>bar</td>
<td>jmpi</td>
<td>jmpi</td>
<td>jump</td>
<td>push</td>
<td>push</td>
</tr>
<tr>
<td>jal</td>
<td>bra</td>
<td>bpt</td>
<td>if</td>
<td>if</td>
<td>loop</td>
<td>push_else</td>
<td>push else</td>
</tr>
<tr>
<td>jr</td>
<td>brk</td>
<td>else</td>
<td>else</td>
<td>else</td>
<td>endloop</td>
<td>pop</td>
<td>pop</td>
</tr>
<tr>
<td>syscall</td>
<td>brkpt</td>
<td>endif</td>
<td>endif</td>
<td>endif</td>
<td>rep</td>
<td>else_wqm</td>
<td>push_wqm</td>
</tr>
<tr>
<td>syscall</td>
<td>cal</td>
<td>do</td>
<td>while</td>
<td>break</td>
<td>breakloop</td>
<td>loop_start</td>
<td>loop_start_no_al</td>
</tr>
<tr>
<td></td>
<td>cont</td>
<td>while</td>
<td>break</td>
<td>cont</td>
<td>breakrep</td>
<td>loop_start_dx10</td>
<td>loop_start_dx10</td>
</tr>
<tr>
<td></td>
<td>kil</td>
<td></td>
<td></td>
<td>break</td>
<td>continue</td>
<td>loop_end</td>
<td>loop_end</td>
</tr>
<tr>
<td></td>
<td>pbk</td>
<td></td>
<td></td>
<td>cont</td>
<td></td>
<td>loop_continue</td>
<td>loop_continue</td>
</tr>
<tr>
<td></td>
<td>pret</td>
<td></td>
<td></td>
<td>halt</td>
<td>call</td>
<td>loop_break</td>
<td>loop_break</td>
</tr>
<tr>
<td></td>
<td>ret</td>
<td></td>
<td></td>
<td>call</td>
<td></td>
<td>jump</td>
<td>jump</td>
</tr>
<tr>
<td></td>
<td>sisy</td>
<td></td>
<td></td>
<td>return</td>
<td></td>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>trap</td>
<td></td>
<td></td>
<td>return</td>
<td></td>
<td>call_fs</td>
<td>call</td>
</tr>
<tr>
<td></td>
<td>s.s</td>
<td></td>
<td></td>
<td>return_fs</td>
<td></td>
<td>call_fs</td>
<td>call_fs</td>
</tr>
<tr>
<td></td>
<td>bar</td>
<td>bar</td>
<td>jmpi</td>
<td>jmpi</td>
<td>jump</td>
<td>push</td>
<td>push</td>
</tr>
<tr>
<td></td>
<td>bpt</td>
<td>if</td>
<td>if</td>
<td>if</td>
<td>loop</td>
<td>push_else</td>
<td>push_else</td>
</tr>
<tr>
<td></td>
<td>else</td>
<td>else</td>
<td>else</td>
<td>else</td>
<td>endloop</td>
<td>pop</td>
<td>pop</td>
</tr>
<tr>
<td></td>
<td>endif</td>
<td>endif</td>
<td>endif</td>
<td>endif</td>
<td>rep</td>
<td>else_wqm</td>
<td>push_wqm</td>
</tr>
<tr>
<td></td>
<td>do</td>
<td>do</td>
<td>while</td>
<td>break</td>
<td>breakloop</td>
<td>loop_start</td>
<td>loop_start_no_al</td>
</tr>
<tr>
<td></td>
<td>while</td>
<td>while</td>
<td>break</td>
<td>cont</td>
<td>breakrep</td>
<td>loop_start_dx10</td>
<td>loop_start_dx10</td>
</tr>
<tr>
<td></td>
<td>break</td>
<td>break</td>
<td>cont</td>
<td>halt</td>
<td>continue</td>
<td>loop_end</td>
<td>loop_end</td>
</tr>
<tr>
<td></td>
<td>cont</td>
<td>cont</td>
<td></td>
<td>call</td>
<td></td>
<td>loop_continue</td>
<td>loop_continue</td>
</tr>
<tr>
<td></td>
<td>kil</td>
<td>kil</td>
<td></td>
<td>return</td>
<td></td>
<td>loop_break</td>
<td>loop_break</td>
</tr>
<tr>
<td></td>
<td>pbk</td>
<td>pbk</td>
<td></td>
<td>return</td>
<td></td>
<td>jump</td>
<td>jump</td>
</tr>
<tr>
<td></td>
<td>pcnt</td>
<td>pcnt</td>
<td></td>
<td>return</td>
<td></td>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>plongjmp</td>
<td>plongjmp</td>
<td></td>
<td>return</td>
<td></td>
<td>call_fs</td>
<td>call</td>
</tr>
<tr>
<td></td>
<td>pret</td>
<td>pret</td>
<td></td>
<td>return_fs</td>
<td></td>
<td>call_fs</td>
<td>call_fs</td>
</tr>
<tr>
<td></td>
<td>ret</td>
<td>ret</td>
<td></td>
<td>return_fs</td>
<td></td>
<td>return</td>
<td>return</td>
</tr>
<tr>
<td></td>
<td>sisy</td>
<td>sisy</td>
<td></td>
<td>return</td>
<td></td>
<td>alu</td>
<td>alu</td>
</tr>
<tr>
<td></td>
<td>s.s</td>
<td>s.s</td>
<td></td>
<td>return</td>
<td></td>
<td>alu_push_before</td>
<td>alu_push_before</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>alu_pop_after</td>
<td>alu_pop_after</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>alu_pop2_after</td>
<td>alu_pop2_after</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>alu_continue</td>
<td>alu_continue</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>alu_break</td>
<td>alu_break</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>alu_else_after</td>
<td>alu_else_after</td>
</tr>
</tbody>
</table>

Control instructions in some CPU and GPU instruction sets

- Why so many?
  - Expose control flow **structure** to the instruction sequencer
- No generic support for arbitrary control flow
Alternative: 1 PC / thread

Code

```cpp
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>
<tbody>
<tr>
<td>PC0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Match → active

No match → inactive
Scheduling policy: min(SP:PC)

Which PC to choose as master PC?

- Conditionals, loops
  - Order of code addresses
  - min(PC)
- Functions
  - Favor max nesting depth
  - min(SP)
- With compiler support
  - Unstructured control flow too
  - No code duplication

Source

```c
if(...) {
  ... p? br else
}
else {
  ... br endif
else:
  ... endif:

while(...) start:
{
  ... p? br start
}
```

Assembleur

```c
while(...) start:
{
  ... p? br start
}
```

Ordre

1. if(...)
2. else
3. else:
4. while(...)
5. while(...) start:
6. while(...) start:
7. while(...) start:
8. while(...) start:
9. while(...) start:
10. while(...) start:
11. while(...) start:
12. while(...) start:
13. while(...) start:
14. while(...) start:
15. while(...) start:
16. while(...) start:

Our new SIMT pipeline

Benefits of multiple-PC arbitration

- Before: stack, counters
  - \( O(d), O(\log d) \) memory
    \( d = \) nesting depth
  - C-style **structured** control-flow only
  - 1 R/W port to memory
  - **Exceptions**: stack overflow, underflow

- Partial SIMD semantics (Bougé-Levaire)
  - Structured control flow only
  - Specific instruction sets

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

- Full SPMD semantics (multi-thread)
  - Traditional languages, compilers
  - Traditional instruction sets
  - **Enables many new architecture ideas**
Outline

- SIMT architectures
- How to keep threads synchronized
- Two instructions, multiple data
  - From divergent branches
  - From multiple warps
- Parallel value locality
# Sharing 2 resources

<table>
<thead>
<tr>
<th>Resource type:</th>
<th>Instruction Fetch</th>
<th>Computation / registers (Data)</th>
<th>Memory port (Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource count</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T_0 T_1 T_2 T_3</td>
<td>T_0 T_1 T_2 T_3</td>
<td>T_0 T_1 T_2 T_3</td>
</tr>
<tr>
<td></td>
<td>F SIMT</td>
<td>X SDMT</td>
<td>M SAMT</td>
</tr>
<tr>
<td>2</td>
<td>T_0 T_1 T_2 T_3</td>
<td>T_0 T_1 T_2 T_3</td>
<td>T_0 T_1 T_2 T_3</td>
</tr>
<tr>
<td></td>
<td>F F DIMT</td>
<td>X X DDMT</td>
<td>M M DAMT</td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>F F F MIMT</td>
<td>X X X MDMT</td>
<td>M MMMM MAMT</td>
</tr>
</tbody>
</table>

Simultaneous Branch Interweaving

- Co-issue instructions from divergent branches
  - Fill holes using parallelism from divergent paths

Control-flow graph

![Diagram showing SIMT and SBI scheduling]

SIMT (baseline)

<table>
<thead>
<tr>
<th>Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2:6</td>
</tr>
<tr>
<td>W1:6</td>
</tr>
<tr>
<td>W2:5</td>
</tr>
<tr>
<td>W1:5</td>
</tr>
<tr>
<td>W2:4</td>
</tr>
<tr>
<td>W1:4</td>
</tr>
<tr>
<td>W2:3</td>
</tr>
<tr>
<td>W1:3</td>
</tr>
<tr>
<td>W2:2</td>
</tr>
<tr>
<td>W1:2</td>
</tr>
<tr>
<td>W2:1</td>
</tr>
<tr>
<td>W1:1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

SBI

<table>
<thead>
<tr>
<th>Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2:6</td>
</tr>
<tr>
<td>W1:6</td>
</tr>
<tr>
<td>W2:4</td>
</tr>
<tr>
<td>W1:4</td>
</tr>
<tr>
<td>W2:3</td>
</tr>
<tr>
<td>W1:3</td>
</tr>
<tr>
<td>W2:2</td>
</tr>
<tr>
<td>W1:2</td>
</tr>
<tr>
<td>W2:1</td>
</tr>
<tr>
<td>W1:1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Secondary scheduler

- W2:5
- W1:5
Secondary scheduler policy

- Primary scheduler: $\text{MPC}_1 = \text{Min}_i (\text{PC}_i)$
- Secondary scheduler: $\text{MPC}_2 = \text{Min}_i (\text{PC}_i, \text{PC}_i \neq \text{MPC}_1)$
- Enforce control-flow reconvergence
  - Annotate reconvergence points with pointer to dominator
  - Wait for any thread of the warp between PCdiv and PCrec

![Diagram showing control-flow reconvergence]

- T0 and T2 (at F) wait for T1 (in D).
- T3 (in B) can proceed in parallel.
**Implementation**

- Fermi GPUs already have 2 instruction schedulers
  - Direct both schedulers to the same units

Fermi: warp size 32
2 warps / clock
1 instruction / warp

SBI: warp size 64
1 warp / clock
2 instructions / warp
Simultaneous Warp Interweaving

- Co-issue instructions from different warps

Transposition of Simultaneous Multi-Threaded (SMT) in the SIMD world
Implementation: cascaded scheduling

- Secondary scheduler refines initial scheduling
  - Looks for warp instruction with disjoint set of active threads

SBI/SWI: warp size 64
1 warp / clock
Detecting compatible warps

- Bitset inclusion test:
  - Content-Associative Memory
    - Treat zeros as don't care bits
    - Power-hungry!

- Set-associative lookup
  - Split warps in sets
  - Restrict lookup to 1 set
  - More power-efficient
Set-associative lookup is good enough

- 3-way: 97% of fully-associative (23-way) performance
- Direct-mapped: 96%
Using divergence correlations

- Issue: unbalanced divergence introduces conflicts
  - e.g. Parallel reduction

- Solution: static lane shuffling
  - Apply different lane permutation for each warp
  - Preserves inter-thread memory locality
Collaboration with Nicolas Brunie (LIP, ENS Lyon / Kalray), Gregory Diamos (Georgia Tech / NVIDIA)
Outline

- SIMT architectures
- How to keep threads synchronized
- Two instructions, multiple data
- Parallel value locality
  - Dynamic scalarization
  - Affine vector cache
  - Affine-aware register allocation
32 birds with 1 stone

- What about SISDSAMT?
  - Phenomenon: parallel **value** locality
  - Applications: instruction sharing, register sharing

- Not as crazy as it looks...
What are we computing on?

- **Uniform data**
  - In a warp, \(v[tid] = c\)

- **Affine data**
  - In a warp, \(v[tid] = b + tid \times s\)
  - Base \(b\), stride \(s\)

- **Average frequency in GPGPU applications**
Dynamic scalarization: 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 scalarization: clock-gating

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
Why on-chip memory size matters

- Conventional wisdom
  - Cache area in CPU vs. GPU according to the NVIDIA CUDA Programming Guide:
    - NVIDIA GF110: 3.9 MB
    - AMD Cayman: 7.7 MB

- 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…
What is inside thread-private memory?

- Private memory: extension to the RF
  - Contains call stack, local arrays, spilled registers

- 80% of private memory traffic is affine
  - RF traffic was 50%
Affine Vector Cache

- As level-1 cache
  - Private memory physically interleaved across threads
    1 cache line = 1 spilled vector register
  - Affine vectors: store \((\text{base}, \text{stride})\) only

Research project of Alexandre Kouyoumdjian, LIP, ENS Lyon, April-May 2011
What is inside a GPU register file?

• Non-affine registers alive in inner loop:
  - MatrixMul: 3 non-affine / 14
  - Convolution: 4 non-affine in hotspot / 14
  - Needleman-Wunsch: 2 non-affine / 24

• 50% - 92% of GPU RF contains affine variables
  ‣ More than register reads: non-affine variables are short-lived
  ‣ Also explains private memory traffic

Research project of Élie Gédéon, LIP, ENS Lyon, June-July 2011
Compilers to the rescue

- Static analysis to identify affine registers
- Issue: divergent control-flow introduces dependencies
  - Solution: gated-SSA form + live-range splitting
- Application: spill affine variables to shared memory

- Up to 40% speedup on current GPUs, for 8 registers / thread

Collaboration with Fernando Magno Quintão Pereira, Diogo Sampaio, Rafael Martins, Universidade Federal de Minas Gerais, Brazil
Future direction: affine cache as RF

- 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 replacement policy?
Bottom line: the missing link

- New micro-architecture space between Clustered Multi-Threading and SIMD
- New ways to exploit parallel value locality for higher perf/W
Conclusion: research factorization?

- Clustered multi-thread architectures: choose between
  - Replication
  - Time-multiplexing
  - Factorization New!

- Instruction fetch policy in multi-thread processors: balance
  - Instruction throughput
  - Fairness
  - Parallel locality New!

- Control-flow reconvergence points
  - For latency: to reduce branch misprediction penalty
  - For throughput: to restore thread synchronization New!

- Cross-fertilization with ideas from “classical” superscalar microarchitecture?
Escaping the SIMD vs. MIMD mindset
a new class of hybrid microarchitectures
between GPUs and CPUs

Sylvain Collange
Università degli Studi di Siena
sylvain.collange@gmail.com

Séminaire DALI
December 15, 2011