Single-Instruction, Single-Data, Multiple-Thread
How to kill 32 birds with 1 stone

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

Groupe de travail Arénaire
December 8, 2011
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
  - Programming model: many-thread, single-kernel
Outline

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

- Application behavior likely to follow regular patterns

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

<table>
<thead>
<tr>
<th>Control regularity</th>
<th>Memory locality, regularity</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Regular/local case</th>
<th>Irregular case</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0</td>
<td>taken</td>
<td>not tk</td>
</tr>
<tr>
<td>i=1</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>i=2</td>
<td>taken</td>
<td>taken</td>
</tr>
<tr>
<td>i=3</td>
<td>taken</td>
<td>not tk</td>
</tr>
</tbody>
</table>

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

**Applications**

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

- Similarity in behavior between threads

<table>
<thead>
<tr>
<th>Parallel control regularity</th>
<th>Regular</th>
<th>Irregular</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>i=17</td>
<td>i=21</td>
</tr>
<tr>
<td>Thread 2</td>
<td>i=17</td>
<td>i=4</td>
</tr>
<tr>
<td>Thread 3</td>
<td>i=17</td>
<td>i=17</td>
</tr>
<tr>
<td>Thread 4</td>
<td>i=17</td>
<td>i=2</td>
</tr>
</tbody>
</table>

```
switch(i) {
    case 2:...
    case 17:...
    case 21:...
}
```

<table>
<thead>
<tr>
<th>Parallel memory locality</th>
<th>Regular</th>
<th>Irregular</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Memory</td>
<td>Memory</td>
</tr>
<tr>
<td>r=A[i]</td>
<td></td>
<td>r=A[i]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parallel value locality</th>
<th>Regular</th>
<th>Irregular</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=32</td>
<td>a=32</td>
<td>a=17</td>
</tr>
<tr>
<td>a=32</td>
<td>a=17</td>
<td>a=5</td>
</tr>
<tr>
<td>a=32</td>
<td>a=5</td>
<td>a=11</td>
</tr>
<tr>
<td>a=32</td>
<td>a=11</td>
<td>a=42</td>
</tr>
<tr>
<td>b=52</td>
<td>b=52</td>
<td>b=15</td>
</tr>
<tr>
<td>b=52</td>
<td>b=15</td>
<td>b=0</td>
</tr>
<tr>
<td>b=52</td>
<td>b=0</td>
<td>b=-2</td>
</tr>
<tr>
<td>r=a*b</td>
<td></td>
<td>r=a*b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b=52</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)

- **Cooperative sharing**
  - If we have parallel locality
  - Same resource, **same** time
  - Single-Instruction Multi-Threading (SIMT)
**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

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

- 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
- Toward value-aware arithmetic?
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

1111

1111 1100
tid=0  tid=2
tid=1  tid=3

1111 1100 1000

1111 1100

1111 1100 0100

1111 1100

1111
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

```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>
<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 | Assembleur | Ordre
--- | --- | ---
if(...) ... p? br else
{
}
else br endif else:
{
}
endif:

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

... f(); call f

void f()
{

... ret

Our new SIMT pipeline

S. Collange. *Une architecture unifiée pour traiter la divergence de contrôle et la divergence mémoire en SIMT.* SympA’14, 2011.
Benefits of multiple-PC arbitration

- Before: stack, counters
  - $O(d)$, $O(\log d)$ memory
    $d = \text{nesting depth}$
  - C-style \textbf{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
  - \textbf{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
- Toward value-aware arithmetic?
Sharing 2 resources

Resource type: Instruction
Resource count: 1

- **Instruction Fetch**

  - **Computation / registers (Data)**
    - **Memory port (Address)**

  - **Resource type:** Memory port
    - **Resource count:** 1

  - **SIMT**
    - **SDMT**
    - **SAMT**

- **Resource type:** Computation / registers (Data)
  - **Resource count:** 2

  - **DIMT**
    - **DDMT**
    - **DAMT**

  - **Resource type:** Memory port (Address)
    - **Resource count:** 1

  - **MIMT**
    - **MMDT**
    - **MAMT**

Simultaneous Branch Interweaving

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

![Control-flow graph and ALUs comparison between SIMT (baseline) and SBI](image)

- Same warp, different instruction
- Secondary scheduler
Simultaneous Warp Interweaving

- Co-issue instructions from different warps
  - Transposition of Simultaneous Multi-Threaded (SMT) in the SIMD world

```
Primary scheduler   ALUs   Secondary scheduler
      ↓       ↓               ↓       ↓
  6  6  6  6   -    6  6  6  6   -    6  6  6  6   -
  6  6  6  6   -    6  6  6  6   -    6  6  6  6   -
  5  5  5  5   -    5  5  5  5   -    5  5  5  5   -
  4    4    4    -    4    4    4    -    4    4    4    -
  4  3  3  4   W2:3  4  3  3  4   W2:3  4  3  3  4   W2:3
  3  2  2  3   W2:2  3  2  2  3   W2:2  3  2  2  3   W2:2
  2    2    2    -    2    2    2    -    2    2    2    -
  1  1  1  1   -    1  1  1  1   -    1  1  1  1   -
  1  1  1  1   -    1  1  1  1   -    1  1  1  1   -

Different warp, different instruction

Primary scheduler   ALUs   Secondary scheduler
      ↓       ↓               ↓       ↓
W2:6  W1:6  W1:4  W1:3  W1:2  W2:1  W1:1
  6  6  6  6   -    6  6  6  6   -    6  6  6  6   -
  6  6  6  6   -    6  6  6  6   -    6  6  6  6   -
  3  3  3  3   W2:3  3  3  3  3   W2:3  3  3  3  3   W2:3
  2  5  5  2   W2:2  2  5  5  2   W2:2  2  5  5  2   W2:2
  1  1  1  1   W2:1  1  1  1  1   W2:1  1  1  1  1   W2:1
  1  1  1  1   W1:1  1  1  1  1   W1:1  1  1  1  1   W1:1

SWI   SBI+SWI
```
Implementation of SBI/SWI

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

Fermi: warp size 32
2 warps / clock

SBI/SWI: warp size 64
1 warp / clock
Results

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
- Toward value-aware arithmetic?
32 birds with 1 stone

- What about SISDSAMT?
  - Instruction 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**

```
thread 1
thread 0
5 5 5 5 5 5 5 5

8 9 10 11 12 13 14 15
```

```
c=5
s=1
b=8
```
Dynamic scalarization: tagging registers

- Associate a tag to each vector register
  - Uniform, Affine, unkown

- Propagate tags across arithmetic instructions

- 2 lanes are enough to encode uniform and affine vectors

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

- Tagging

<table>
<thead>
<tr>
<th>Tag</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>0 1 2 3 ...</td>
</tr>
<tr>
<td>U</td>
<td>t i n</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:

- 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
  - Extension to the RF: 1 cache line = 1 spilled vector register
  - Space-efficient storage of affine vectors

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?
SIMT architectures
How to keep threads synchronized
Two instructions, multiple data
Parallel value locality
Toward value-aware arithmetic?
Partial parallel value locality

Work in progress

- p-bit SDMT vector
  - All vector components have identical $p$ high-order bits

- p-bit DDMT vector
  - 2 different values of $p$ high-order bits for the whole vector
Preliminary results: Integer RF reads

- 84% of operands are 24-bit SDMT
- 94% are 24-bit DDMT or SDMT
  - 100% in 23 kernels / 39
Mixed scalar-vector arithmetic

- Factor out high-order bits
  \( n \times 64\text{-bit} \rightarrow 64\text{-bit} + n \times 8\text{-bit} \)
  - Overlap for delayed carry propagation (normalization)

- Detecting overflows in multiplies, shifts...
  - After operation
  - Before operation: maintain sideband bit count, use it for instruction scheduling

- Intuition: this is a vector address generation unit
  - 64-bit pointer + 8-bit offsets
Floating-point data

- Less than integer, but still significant
  - Wide variations
  - >50% 24-bit DDMT on average
  - Caveat: many benchmarks compute on random values!
- Toward value-aware arithmetic?
Conclusion: the missing link

- New micro-architecture space between Clustered Multi-Threading and SIMD
- New ways to exploit parallel value locality for higher perf/W
Single-Instruction, Single-Data, Multiple-Thread
How to kill 32 birds with 1 stone

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

Groupe de travail Arénaire
December 8, 2011

Disclaimer: No animal was harmed during the writing of these slides.