Multi-threading or SIMD?  
How GPU architectures exploit regularity

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

ARCHI'11  
June 14, 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
  - 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
  - Dealing with variable latency
The 1980's: pipelined processor

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

```
for i = 0 to n-1
    X[i] ← a \times X[i]
```

Source code

```
move i ← 0
loop:
    load t ← X[i]
    mul t ← a\times t
    store X[i] ← t
    add i ← i+1
    branch i<n? loop
```

Machine code

Sequential CPU

- Fetch
- Decode
- Execute
- L/S Unit
- Memory
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

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

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

```
j=17 | j=18 | j=19 | j=20 |
```

- Applications
  - Caches
  - Branch prediction
  - Instruction prefetch, data prefetch, write combining…
The 2000s: 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

```plaintext
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:  

- Improves throughput thanks to explicit parallelism
Interleaved multi-threading

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

```
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
  - Dealing with variable latency
Heterogeneity: causes and consequences

- **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**
  - Power-constrained: can afford idle transistors
  - Suggests more radical specialization
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></td>
<td></td>
</tr>
<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>

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

<table>
<thead>
<tr>
<th><strong>Memory regularity</strong></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Memory</td>
<td>r=A[i]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Data regularity</strong></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a=32</td>
<td>a=17</td>
<td>a=17</td>
</tr>
<tr>
<td>a=32</td>
<td>a=-5</td>
<td>a=11</td>
</tr>
<tr>
<td>a=32</td>
<td>a=42</td>
<td>a=42</td>
</tr>
<tr>
<td>b=52</td>
<td>b=15</td>
<td>b=-2</td>
</tr>
<tr>
<td>b=52</td>
<td>b=0</td>
<td>b=52</td>
</tr>
<tr>
<td>b=52</td>
<td>b=0</td>
<td>b=52</td>
</tr>
<tr>
<td>r=a*b</td>
<td>r=a*b</td>
<td>r=a*b</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
  - Dealing with variable latency
Capturing instruction regularity

- How to handle control divergence?
  - Techniques from Single Instruction, Multiple Data (SIMD) architectures

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

```plaintext
x = 0;
// Uniform condition
if(tid > 17) {
    x = 1;
}

// Divergent conditions
if(tid < 2) {
    if(tid == 0) {
        x = 2;
    } else {
        x = 3;
    }
}
```
Most common: mask stack

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
```

tid=0  tid=2

```
1111
```

tid=1  tid=3

```
1111 1100
```

```
1111 1100 1000
```

```
1111 1100
```

```
1111 1100 0100
```

```
1111 1100
```

```
1111
```

Curiosity: activity counters

Code
\[ x = 0; \]

// Uniform condition
\[
\text{if}(\text{tid} > 17) \{ \\
\quad x = 1; \\
\}
\]

// Divergent conditions
\[
\text{if}(\text{tid} < 2) \{ \\
\quad \text{inc} \quad \text{if}(\text{tid} == 0) \{ \\
\quad\quad \text{inc} \quad x = 2; \\
\quad\quad \text{dec} \\
\quad \} \\
\quad \text{else} \{ \\
\quad\quad \text{inc} \quad x = 3; \\
\quad\quad \text{dec} \\
\quad \} \\
\quad \text{dec}
\}
\]

Counters
1 (in)activity counter / thread

Brute-force: 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>
</table>

- Match → active
- No match → inactive

Traditional SIMT pipeline

- Used in virtually every modern GPU
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>
</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>
</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>
</tr>
<tr>
<td>jr</td>
<td>brk</td>
<td>else</td>
<td>else endif</td>
<td>endloop</td>
<td>pop</td>
<td>pop</td>
</tr>
<tr>
<td>syscall</td>
<td>brkp</td>
<td>endif</td>
<td>endif</td>
<td>rep</td>
<td>push_wqm</td>
<td>push_wqm</td>
</tr>
<tr>
<td>cal</td>
<td>cal</td>
<td>do</td>
<td>while break</td>
<td>endrep</td>
<td>pop_wqm</td>
<td>pop_wqm</td>
</tr>
<tr>
<td>cont</td>
<td>cont</td>
<td>break</td>
<td>cont</td>
<td>breakloop</td>
<td>else_wqm</td>
<td>else_wqm</td>
</tr>
<tr>
<td>kil</td>
<td>exit</td>
<td>msave</td>
<td>call</td>
<td>breakrep</td>
<td>jump_any</td>
<td>jump_any</td>
</tr>
<tr>
<td>pbk</td>
<td>jcal</td>
<td>halt</td>
<td>halt</td>
<td>continue</td>
<td>reactivate</td>
<td>reactivate</td>
</tr>
<tr>
<td>pret</td>
<td>jmx</td>
<td>msave</td>
<td>call return</td>
<td>loop_start</td>
<td>reactivate_wqm</td>
<td>reactivate_wqm</td>
</tr>
<tr>
<td>ret</td>
<td>kil</td>
<td>push</td>
<td>push pop</td>
<td>loop_start_no_al</td>
<td>jump</td>
<td>jump</td>
</tr>
<tr>
<td>ssy</td>
<td>pbk</td>
<td>pop2_after</td>
<td>alu break</td>
<td>loop_start_dx10</td>
<td>call</td>
<td>call</td>
</tr>
<tr>
<td>trap</td>
<td>pret</td>
<td>call continue</td>
<td>alu</td>
<td>loop_end</td>
<td>call_fs</td>
<td>call_fs</td>
</tr>
<tr>
<td>.s</td>
<td>sisy</td>
<td>alu_break</td>
<td>alu</td>
<td>loop_continue</td>
<td>return</td>
<td>return</td>
</tr>
<tr>
<td></td>
<td>.s</td>
<td>alu_else_after</td>
<td>alu</td>
<td>loop_break</td>
<td>return_fs</td>
<td>return_fs</td>
</tr>
</tbody>
</table>

Control instructions in some CPU and GPU instruction sets

- Why so many?
  - Expose control flow **structure** to the instruction sequencer
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: threads which are late
  - Deepest control flow nesting level
  - Deepest function call nesting level
  - Various combinations of the former


Our new SIMT pipeline

Instructions are fetched into the pipeline, matched against pre-computed data, and executed. If there is no match, the instruction is discarded.
Benefits of multiple-PC arbitration

- Before: stack, counters
  - $O(n)$, $O(\log n)$ memory
    $n =$ 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
Dual Instruction, Multiple Threads (DIMT)

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

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

PE

- More than 2 instructions: NIMT

Why DIMT?

- “Fills holes” using parallelism between execution paths.
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

New DIMT+DWF pipeline

- Radical departure from classical SIMD
Avoiding redundancy

- Goal: keep execution units busy?

- Keep execution units busy 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**

![Operations and Inputs Bar Chart]

- **Operations**
  - Other: 70%
  - Affine: 20%
  - Uniform: 10%

- **Inputs**
  - Other: 100%
  - Affine: 100%
  - Uniform: 100%
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

Instructions

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

Tags

```
A←A
K←U[A]
K←U×K
U[A]←K
A←A+U
A<U?
```

Tagging registers

- Thread 0 1 2 3 ...
- Tag K U A U
- Trace

- Trace
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
  - Dealing with variable latency
It's the memory, stupid!

- 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 (most)

Memory access patterns

In traditional vector processing

- **Scalar load & broadcast**
- **Reduction & scalar 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 DRAM 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

- **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
  - Dealing with variable latency
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
  - Could counteract utilization loss with DIMT/NIMT?

Linked list traversal: without DWS

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

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

Thread 0
PC=2
hit

Thread 1
PC=2
miss

Thread 2
PC=2
hit

Thread 3
PC=2
miss

MPC=1
1: i != -1?
true

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

MPC=1
1: i != -1?
true

MPC=1
1: i != -1?
false

PC=3
Linked list traversal: with DWS

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

Thread 0
MPC=2
2: i = l[i];
PC=2
hit
PC=1
true
MPC=1
1: i != -1? 
PC=2
miss
PC=2
hit
PC=3
hit
PC=3
hit
PC=3
true
MPC=1
1: i != -1?
PC=2
true
PC=2
true
Thread 1
PC=2
miss
PC=1
true
MPC=1
1: i != -1?
PC=2
hit
PC=2
hit
PC=2
hit
PC=3
false
MPC=2
2: i = l[i];
PC=2
hit
PC=1
true
PC=1
false
PC=3
true
Thread 2
PC=1
ready=0
PC=1
true
PC=3
PC=3
Thread 3
PC=2
miss
PC=1
true
PC=1
false
PC=3
PC=3
ready=0
SIMT pipeline – memory instruction

Hazards: Divergence, Bank conflict, Cache miss
all cause PC and valid bit to be updated accordingly
Conclusion: the missing link

- New range of architecture options between Simultaneous Multi-Threading, Chip Multi-Threading and SIMD
  - Exploits parallel regularity for higher perf/W
Perspectives: next challenges

- Instruction fetch policy, thread scheduling policy: objectives to balance
  - Instruction throughput
  - Memory-level parallelism
  - Fairness
  - Regularity — coherence

- Detect control-flow reconvergence points

- Cross-fertilization with ideas from “classical” multi-threaded microarchitecture?
Multi-threading or SIMD?
How GPU architectures exploit regularity

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

ARCHI'11
June 14, 2011
Conclusion

- SIMT bridges the gap between superscalar and SIMD
  - Smooth, dynamic tradeoff between regularity and efficiency

- **Efficiency** (flops/W)

- **Superscalar**
  - Dynamic warp formation,
  - Dynamic warp subdivision,
  - NIMT…

- **SIMT**
  - Dynamic work factorization
  - Decoupled scalar-SIMT,
  - Affine caches…

- **SIMD**

- **Regularity of application**
  - Transaction processing
  - Computer graphics
  - Dense linear algebra
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

For $n$ threads:

\[
X[tid] \leftarrow a \times X[tid]
\]
void scale(float a, float * X, int n) {
    for(int i = 0; i != n; ++i)
        X[i] = a * X[i];
}

scale:
    test esi, esi
    je .L4
    sub esi, 1
    xor eax, eax
    lea rdx, [4+rsi*4]
    .L3:
    movss xmm1, DWORD PTR [rdi+rax]
    mulss xmm1, xmm0
    movss DWORD PTR [rdi+rax], xmm1
    add rax, 4
    cmp rax, rdx
    jne .L3
    .L4:
    rep
    ret
GPU microarchitecture

Software

__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
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

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

**Code**

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

**Trace**

- $w_0$: load $X[\{0,1,2,3\}]$
- $w_1$: load $X[\{4,5,6,7\}]$
- $w_2$: load $X[\{8,9,10,11\}]$

- "Horizontal" temporal or spatial locality
  - Parallel reuse, in space
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
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**
- **vmul**
- **vstore**
- **add**
- **branch**

### Registers

- **T**
- **i**
- **a**
- **t**
- **n**
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**

  for $i = 0$ to $n-1$ step 4  
  $X[i..i+3] \leftarrow a \times X[i..i+3]$

  **Source code**

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

  **Machine code**

  **SIMD CPU**

- **Challenging to program (semi-regular apps?)**
So how does it differ from SIMD?

<table>
<thead>
<tr>
<th></th>
<th>SIMD</th>
<th>SIMT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction regularity</td>
<td>Vectorization at compile-time</td>
<td>Vectorization at runtime</td>
</tr>
<tr>
<td>Control regularity</td>
<td>Software-managed Bit-masking, predication</td>
<td>Hardware-managed Stack, counters, multiple PCs</td>
</tr>
<tr>
<td>Memory regularity</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: accuracy

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

![Bar chart showing accuracy of detections for inputs and outputs.](chart.png)