Data regularity for efficient parallel architectures

Sylvain Collange
Arénaire, LIP, ENS de Lyon

Kalray

June 8, 2011
Context

- Accelerate compute-intensive applications
  - HPC: computational fluid dynamics, seismic imaging, DNA folding, phylogenetics…
  - Multimedia: 3D rendering, video, image processing…

- Current constraints
  - Power consumption
  - Cost of moving and retaining data

- Focus on GPU
  - Video game: mass market
    - Economy of scale, amortized R&D
  - High-performance parallel processor
    - For much more than graphics
  - Now in #1 supercomputer (Tianhe-1A)
Why should we care about GPUs?

- **Yesterday (…-2010)**
  - Homogeneous multi-core
  - Discrete components

- **Today (2011-…)**
  Heterogeneous multi-core
  - Intel Sandy Bridge shipping
  - AMD Fusion sampling
  - NVIDIA Denver/Maxwell project announced
Why should we care about GPUs?

- **Yesterday (…-2010)**
  - Homogeneous multi-core
  - Discrete components

- **Today (2011-…)**
  - Heterogeneous multi-core
    - Intel Sandy Bridge shipping
    - AMD Fusion sampling
    - NVIDIA Denver/Maxwell project announced
Outline

- Step-by-step introduction to GPU architecture
  - Software model
  - Exploiting explicit parallelism
  - Exploiting implicit regularity
- Exploiting data regularity
  - Limitations of current GPUs
  - Dynamic data deduplication
  - Static data deduplication
- Ongoing work
- Conclusion
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

- Example: in-place scalar-vector product
  - \[ X \leftarrow a \times X \]

- Let's build a computer to run these programs
1st step: sequential processor

- Sequential execution

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

- Can exploit implicit instruction-level parallelism to increase throughput
- Obstacles: not scalable
  - David Patterson (UCBerkeley): “Power Wall + Memory Wall + ILP Wall = Brick Wall”
2nd step: homogeneous multi-core

- Replication of the complete execution engine
- Software: coarse-grained threading

```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: T0 T1

- Improves throughput thanks to explicit parallelism
3\textsuperscript{rd} step: interleaved multi-threading

- Time-multiplexing of processing units
- Software: coarse-grained or fine-grained threading

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

Machine code (fine-grained threading)

Threads: T0  T1  T2  T3

- Hides latency thanks to explicit parallelism
4th step: clustered multi-core

- For each individual unit, select between
  - Replication
  - Time-multiplexing
- Examples
  - Sun UltraSparc T2
  - AMD Bulldozer
- Area-efficient tradeoff
Regularity

- Similarity in behavior between threads

<table>
<thead>
<tr>
<th>Instruction regularity</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
<th>Thread 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>mul</td>
<td>mul</td>
<td>mul</td>
<td>mul</td>
<td>mul</td>
</tr>
<tr>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
</tr>
</tbody>
</table>

| Time
| mul       | add      | store    | load     |
| load      | mul      | sub      | add      |

<table>
<thead>
<tr>
<th>Control regularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=17</td>
</tr>
<tr>
<td>i=17</td>
</tr>
<tr>
<td>i=17</td>
</tr>
<tr>
<td>i=17</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Memory regularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>load X[8]</td>
</tr>
<tr>
<td>load X[9]</td>
</tr>
<tr>
<td>load X[10]</td>
</tr>
<tr>
<td>load X[11]</td>
</tr>
</tbody>
</table>

Final step: 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
So how does it differ from SIMD?

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

- SIMD is static, SIMT is dynamic
- Similar opposition as VLIW vs. superscalar
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

- Step-by-step introduction to GPU architecture
  - Software model
  - Exploiting explicit parallelism
  - Exploiting implicit regularity
- Exploiting data regularity
  - Limitations of current GPUs
  - Dynamic data deduplication
  - Static data deduplication
- Ongoing work
- Conclusion
Knowing our baseline

- Design and run micro-benchmarks
  - Target NVIDIA Tesla architecture
  - Go far beyond published specifications
  - Understand design decisions

- Run power studies [CDT09]
  - Energy measurements on micro-benchmarks
  - Understand power constraints
Barra

- Functional GPU simulator [CDDP10]
- Modeled after NVIDIA Tesla GPUs
  - Executes native CUDA binaries
  - Reproduces SIMT execution
- Built within the Unisim framework
- Fast
  - Uses SSE instructions and multi-threading
- Accurate
  - Bit-accurate floating-point
  - Produces low-level statistics
  - Allows experimenting with architecture changes

http://gpgpu.univ-perp.fr/index.php/Barra
Primary constraint: power

- Power measurements on NVIDIA GT200 [CDT09]

<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 32-element vector</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

- Need to keep memory traffic low

- Standard solution: caches
On-chip memory

- 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…
The cost of SIMT: register wastage

**SIMD**

\[
\begin{align*}
\text{mov} & \quad i \leftarrow 0 \\
\text{loop:} & \\
\text{vload} & \quad T \leftarrow X[i] \\
\text{vmul} & \quad T \leftarrow a \times T \\
\text{vstore} & \quad X[i] \leftarrow T \\
\text{add} & \quad i \leftarrow i + 16 \\
\text{branch} & \quad i < n? \quad \text{loop}
\end{align*}
\]

**SIMT**

\[
\begin{align*}
\text{mov} & \quad i \leftarrow \text{tid} \\
\text{loop:} & \\
\text{load} & \quad t \leftarrow X[i] \\
\text{mul} & \quad t \leftarrow a \times t \\
\text{store} & \quad X[i] \leftarrow t \\
\text{add} & \quad i \leftarrow i + \text{tcnt} \\
\text{branch} & \quad i < n? \quad \text{loop}
\end{align*}
\]

**Instructions**

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>branch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Registers**

- **T**
  - Scalar: 0, 0, 0, 0, ..., 0
  - Vector: 17, 17, 17, 17, ..., 17

- **a**
  - Scalar: 17, 17, 17, 17, ..., 17
  - Vector: 15, 15, 15, 15, ..., 15

- **i**
  - Scalar: 51, 51, 51, 51, ..., 51
  - Vector: 51, 51, 51, 51, ..., 51
## SIMD vs. SIMT

<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>
<tr>
<td><strong>Data regularity</strong></td>
<td>Scalar registers, scalar instructions</td>
<td><em>Duplicate registers, duplicated ops</em></td>
</tr>
</tbody>
</table>
Uniform and affine vectors

- **Uniform vector**
  - In a warp, \( v[i] = c \)
  - Value does not depend on lane ID

- **Affine vector**
  - In a warp, \( v[i] = b + i \cdot s \)
  - Base \( b \), stride \( s \)
  - Affine relation between value and lane ID

- **Generic vector**: anything else
How many uniform / affine vectors?

- Analysis by simulation with Barra
  - Applications from the CUDA SDK
  - Granularity 16 threads

- 49% of all reads from the register file are affine
- 32% of all instructions compute on affine vectors

- This is what we have “in the wild”
  - How to capture it?
Outline

- Step-by-step introduction to GPU architecture
  - Software model
  - Exploiting explicit parallelism
  - Exploiting implicit regularity
- Exploiting data regularity
  - Limitations of current GPUs
  - Dynamic data deduplication
  - Static data deduplication
- Ongoing work
- Conclusion
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          Tags
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?
```

```
Tag
0 1 2 3 ...
K
U
A
U
```
Dynamic data deduplication: results

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

![Bar charts showing detection rates of inputs and outputs for different types of data.](image-url)
New pipeline

- New deduplication stage
  - In parallel with predication control stage
- Split RF banks into scalar part and vector part
- Fine-grained clock-gating on vector RF and SIMD datapaths
  - Cooperative sharing of execution units
Power savings

 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
# SIMD vs. SIMT

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


Outline

- Step-by-step introduction to GPU architecture
  - Software model
  - Exploiting explicit parallelism
  - Exploiting implicit regularity
- Exploiting data regularity
  - Limitations of current GPUs
  - Dynamic data deduplication
  - Static data deduplication
- Ongoing work
- Conclusion
A scalarizing compiler?

<table>
<thead>
<tr>
<th>Programming models</th>
<th>Sequential programming</th>
<th>SPMD (CUDA, OpenCL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architectures</td>
<td>Model CPU Scalar</td>
<td>Actual CPU Scalar+SIMD</td>
</tr>
</tbody>
</table>

- Scalar-only
- Vector-only
- Traditional compiler
- Vectorizing compiler
- Scalarizing compiler
- GPU SIMT
Still uniform and affine vectors

- Scalar registers
  - Uniform vectors
  - Affine vectors with known stride
- Scalar operations
  - Uniform inputs, uniform outputs
- Uniform branches
  - Uniform conditions
- Vector load/store (vs. gather/scatter)
  - Affine aligned addresses

\[c = a + b\]
\[
\begin{array}{c}
a \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \\
+ \quad + \quad + \quad + \quad + \quad + \\
b \quad 3 \quad 3 \quad 3 \quad 3 \quad 3 \quad 3 \quad 3 \\
c \quad 5 \quad 5 \quad 5 \quad 5 \quad 5 \quad 5 \quad 5
\end{array}
\]

\[
\text{if}(c) \\
\{
\text{else}
\}
\]

\[
x = t[i];
\]

\begin{tikzpicture}[->,>=stealth',auto,swap]
  \node (1) at (0,0) {Memory};
  \node (2) at (2,0) {t[8]};
  \node (3) at (3,0) {t[15]};
  \node (4) at (1,1) {i};
  \node (5) at (1,2) {x};
  \node (6) at (1,3) {8 \ 9 \ 10 \ 11 \ 12 \ 13 \ 14 \ 15};
  \path (4) edge (5)
  \path (5) edge (6)
  \path (6) edge (2)
  \path (6) edge (3);
\end{tikzpicture}
From SPMD to SIMD

- Forward dataflow analysis
- Statically propagate tags in dataflow graph
  - \(\perp, C(v), U, A(s), K\)
  - Propagate value \(v\) of constants, stride \(s\) of affine vectors, and alignment

\[\phi\]

**SPMD**

\[
\begin{align*}
\text{mov} & \quad i0 \leftarrow \text{tid} \\
& \quad A(1) \leftarrow A(1)
\end{align*}
\]

\[
\begin{align*}
\phi & \quad i1 \leftarrow \phi(i0, i2) \quad A(1) \leftarrow \phi(A(1), \perp) \\
\text{load} & \quad t \leftarrow X[i1] \quad K \leftarrow U[A(1)] \\
\text{mul} & \quad t \leftarrow a \times t \quad K \leftarrow U \times K \\
\text{store} & \quad X[i1] \leftarrow t \quad U[A(1)] \leftarrow K \\
\text{add} & \quad i2 \leftarrow i1 + \text{tcnt} \quad A(1) \leftarrow A(1) + C(\text{tcnt}) \\
\text{branch} & \quad i2 < n? \quad \text{loop} \quad A(1) < C(n)?
\end{align*}
\]
From SPMD to SIMD

- Forward dataflow analysis
- Statically propagate tags in dataflow graph
  - ⊥, C(v), U, A(s), K
  - Propagate value v of constants, stride s of affine vectors, and alignment

SPMD

```
mov i0 ← tid  A(1) ← A(1)
phi i1 ← φ(i0, i2)  A(1) ← φ(A(1), A(1))
load t ← X[i1]  K ← U[A(1)]
mul t ← a×t  K ← U×K
store X[i1] ← t  U[A(1)] ← K
add i2 ← i1+tcnt  A(1) ← A(1)+C(tcnt)
branch i2<n? loop  A(1) ← C(n)?
```

SIMD

```
mov i0 ← 0
phi i1 ← φ(i0, i2)
vload t ← X[i1]
vmul t ← a×t
vstore X[i1] ← t
add i2 ← i1+tcnt
branch i2<n? loop
```

Results: instructions, registers

- Benchmarks: CUDA SDK, SGEMM, Rodinia, Parboil
- Static operands

![Bar chart showing inputs identified and total inputs with categories: Unknown, Affine, Uniform.]

- Split register allocation

![Bar chart showing registers with categories: Vector, Scalar.]

35
- **SIMD**: identify at compile-time situations that SIMT detects at runtime
  - Uniform branches
  - Uniform, unit-strided loads & stores
  - Scalar instructions, registers
## Static vs. Dynamic data deduplication

<table>
<thead>
<tr>
<th>Static deduplication</th>
<th>Dynamic deduplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allows simpler hardware</td>
<td>Preserves binary compatibility</td>
</tr>
<tr>
<td>Governs register allocation and instruction scheduling</td>
<td>Captures dynamic behavior</td>
</tr>
<tr>
<td>Enables constant propagation</td>
<td>Unaffected by (future) software complexity</td>
</tr>
<tr>
<td>Applies to memory (call stack...)</td>
<td></td>
</tr>
</tbody>
</table>
Outline

- Step-by-step introduction to GPU architecture
  - Software model
  - Exploiting explicit parallelism
  - Exploiting implicit regularity
- Exploiting data regularity
  - Limitations of current GPUs
  - Dynamic data deduplication
  - Static data deduplication
- Ongoing work
- Conclusion
My register file is full of holes!

- Static partitioning of RF between warps is space-inefficient

- 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]
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?
Bottom line

- Introduced regularity: instruction, control, memory, data
- Current GPGPU applications exhibit significant data regularity
- Both static and dynamic techniques can exploit it
  - Enables power savings
  - Less data storage and movement
Future challenges

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

![Diagram showing the relationship between regularity of application and efficiency](image)

- Superscalar
- SIMT
- SIMD

- Dynamic warp formation [Fung07], Dynamic warp subdivision [Meng10], NIMT [Glew09]...
- Dynamic deduplication, Decoupled scalar-SIMT, Affine caches?
References


- [CDZ09] Sylvain Collange, David Defour, Yao Zhang. *Dynamic detection of uniform and affine vectors in GPGPU computations*. Europar HPPC09, 2009

