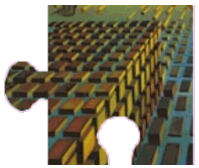
Challenges in GPGPU architectures: fixed-function units and regularity

Sylvain Collange

CARAMEL Seminar
December 9, 2010



DALI
Digits, Architectures et Logiciels Informatiques

UPVD
Université de Perpignan Via Domitia

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 GPGPU

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

Credits: NVIDIA

Outline of this talk

- Introduction to GPU architecture
- Balancing specialization and genericity
  - Current challenges
  - GPGPU using specialized units
- Exploiting regularity
  - Limitations of current GPUs
  - Dynamic data deduplication
  - Static data deduplication
- Conclusion

Sequential processor

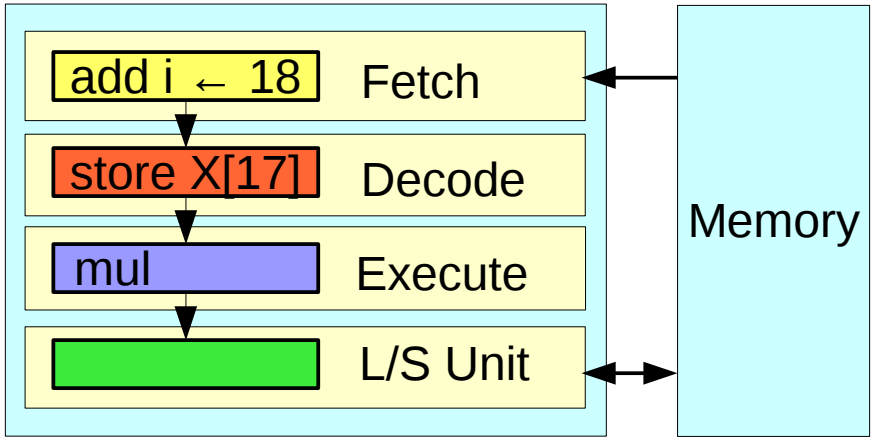
- Example: scalar-vector multiplication: X ← a·X

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

Source code

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

Machine code



Sequential CPU

Sequential processor

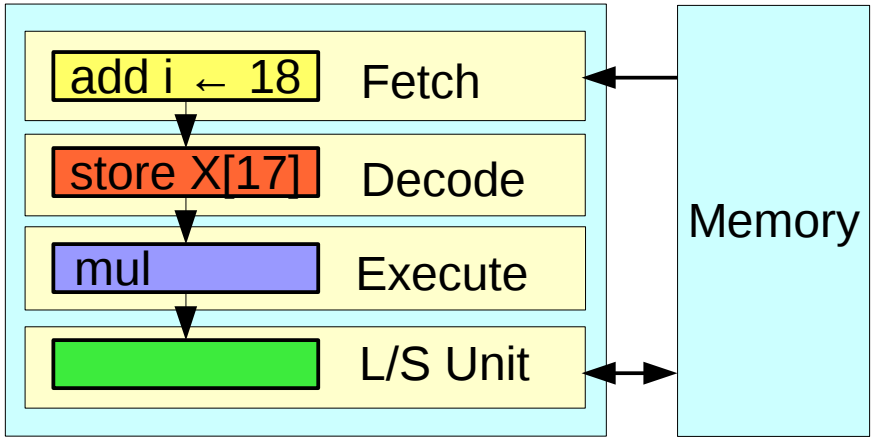
- Example: scalar-vector multiplication: X ← a·X

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

Source code

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

Machine code



Sequential CPU

- Obstacles to increasing sequential CPU performance
  - David Patterson (UCBerkeley):
    “Power Wall + Memory Wall + ILP Wall = Brick Wall”

Multi-core

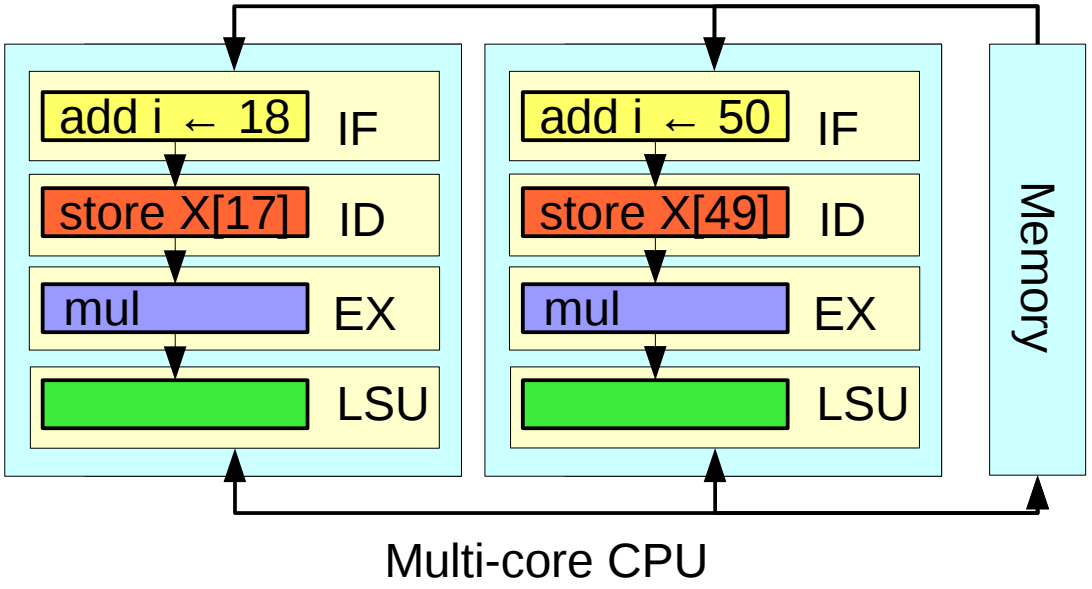
- Break computation into *m* independent *threads*
- Run threads on independent cores

```
for i = kn/m to (k+1)n/m-1
    X[i] ← a * X[i]
```

Source code (thread **k**)

```
    move  i ← kn/m
loop:
    load  t ← X[i]
    mul   t ← a×t
    store X[i] ← t
    add   i ← i+1
    branch i<(k+1)n/m? loop
```

Machine code

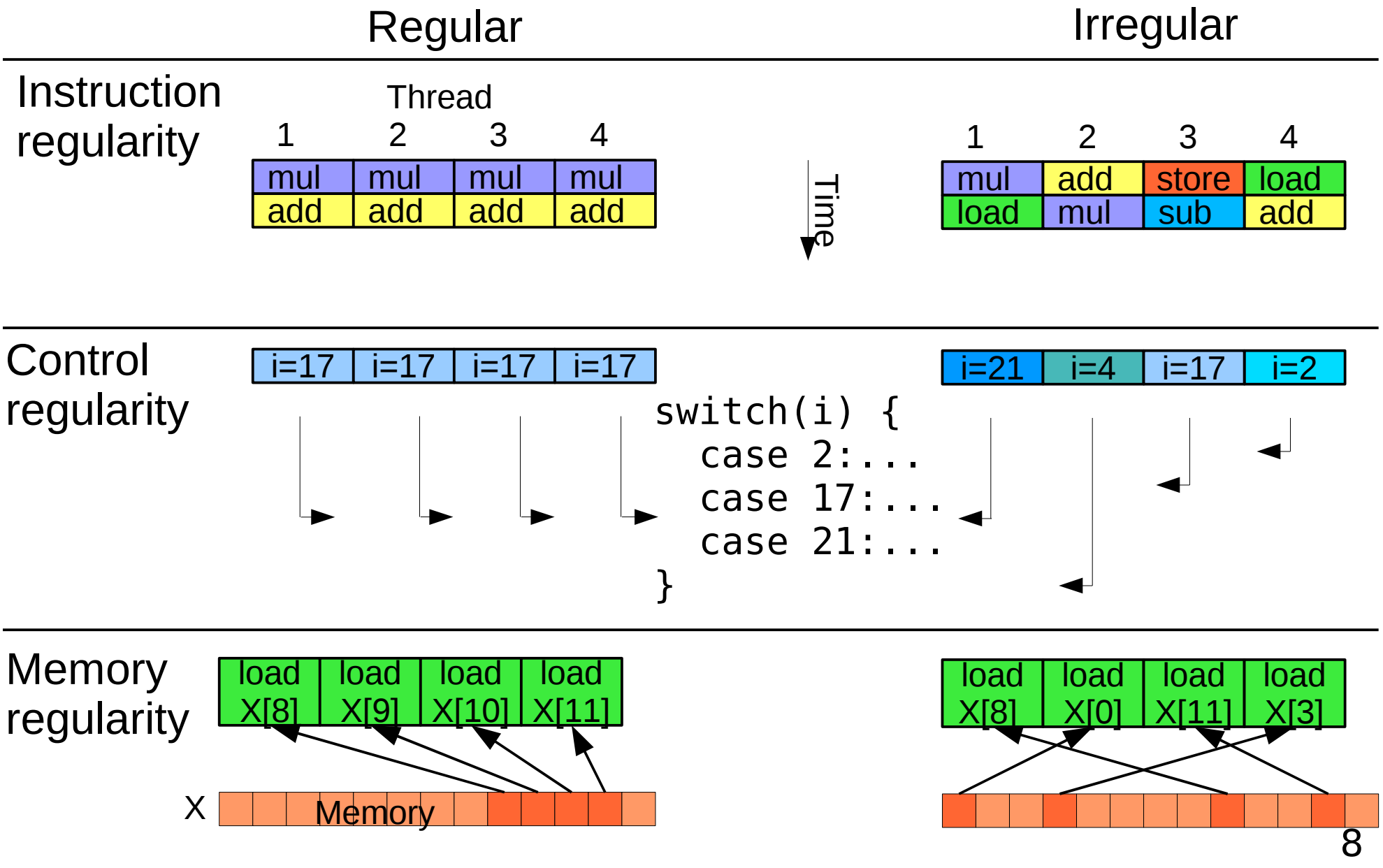


Multi-core CPU

- Benefit from data parallelism

Regularity

- Similarity in behavior between threads



SIMD

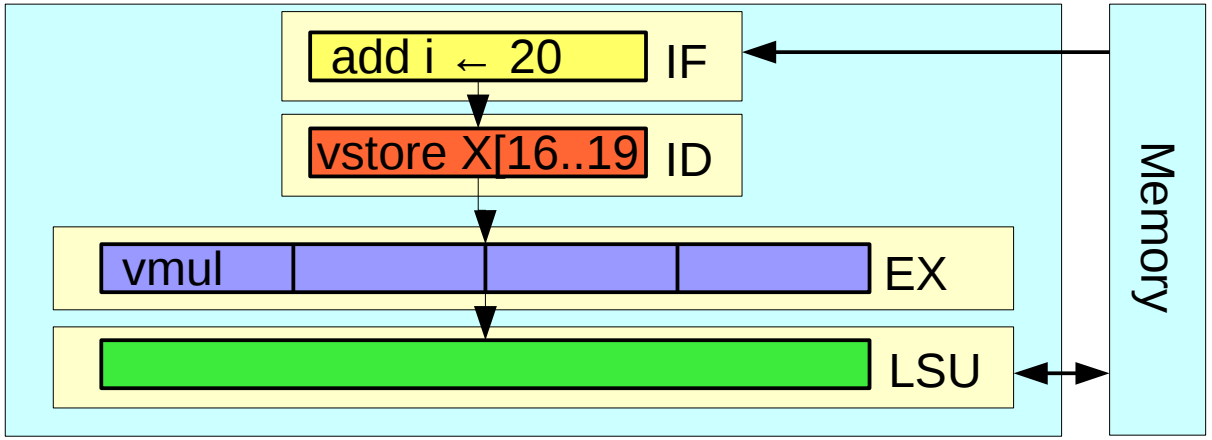
- Single Instruction Multiple Data

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

- Benefit from regularity
- Challenging to program (semi-regular apps?)

SIMT

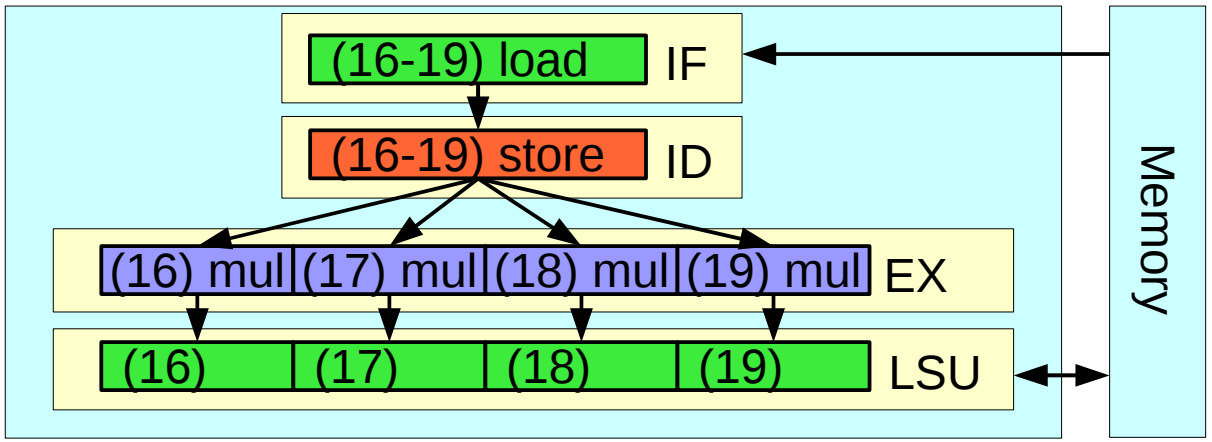
- Single Instruction, Multiple Threads

```
For n threads:
   X[tid] ← a * X[tid]
```

Source code

```
load  t ← X[tid]
mul   t ← a×t
store X[tid] ← t
```

Machine code



SIMT GPU

- Vectorization at runtime
- Group of synchronized threads: *warp*

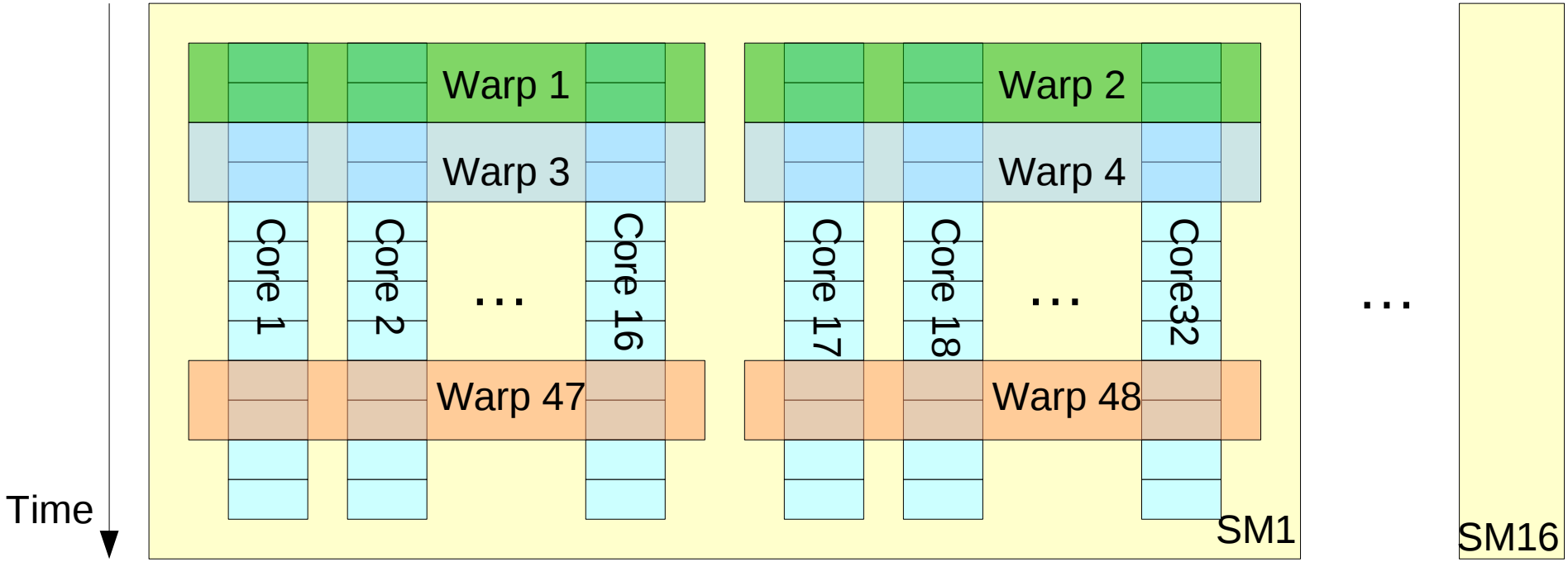
SIMD vs. SIMT

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

- Static vs. dynamic
- Similar contrast 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 of this talk

- Introduction to GPU architecture
- Balancing specialization and genericity
  - Current challenges
  - GPGPU using specialized units
- Exploiting regularity
  - Limitations of current GPUs
  - Dynamic data deduplication
  - Static data deduplication
- Conclusion

2005-2009: the road to unification?

- Example: standardization of arithmetic units
  - 2005: exotic “Cray-1-like” floating-point arithmetic
  - 2007: minimal subset of IEEE 754
  - 2010: full IEEE 754-2008 support
- Other examples of unification
  - Memory access
  - Programming language facilities
- GPU becoming a standard processor
  - Tim Sweeney (EPIC Games): “The End of the GPU Roadmap”
- Intel Larrabee project
  - Multi-core, SIMD CPU for graphics

S. Collange, M. Daumas, D. Defour. État de l'intégration de la virgule flottante dans les processeurs graphiques. *RSTI – TSI 27/2008, p. 719 – 733.* 2008

2010: back to specialization

- 2009-12: Intel Larrabee canceled
  - …as a graphics product
- Specialized units are still alive and well
  - Power efficiency advantage
  - Rise of the mobile market
- Long-term direction
  - Heterogeneous multi-core
  - Application-specific accelerators
- Relevance for HPC?
- Right balance between specialization and genericity?

Contributions of this part

- Radiative transfer simulation in OpenGL
  - >50× speedup over CPU
  - Thanks to specialized units : rasterizer, blending, transcendentals
- Piecewise polynomial evaluation
  - +60% over Horner rule on GPU
  - Creative use of the texture filtering unit
- Interval arithmetic library
  - 120× speedup over CPU
  - Thanks to static rounding attributes

S. Collange, M. Daumas, D. Defour. Graphic processors to speed-up simulations for the design of high performance solar receptors. *ASAP18*, 2007.
S. Collange, M. Daumas, D. Defour. Line-by-line spectroscopic simulations on graphics processing units. *Computer Physics Communications*, 2008.
S. Collange, J. Flòrez, D. Defour. A GPU interval library based on Boost.Interval. *RNC*, 2008.
M. Arnold, S. Collange, D. Defour. Implementing LNS using filtering units of GPUs. *ICASSP*, 2010.
Interval code sample, *NVIDIA CUDA SDK 3.2*, 2010

Beyond GPGPU programming

- Limitations encountered
  - Software: drivers, compiler
    - No access to attribute interpolator in CUDA
  - Hardware: usage scenario not considered at design time
    - Accuracy limitations in blending units, texture filtering
- Broaden application space without compromising (too much) power advantage?
  - GPU vendors willing to include non-graphics features, unless prohibitively expensive
- We need to study GPU architecture

Outline of this talk

- Introduction to GPU architecture
- Balancing specialization and genericity
  - Current challenges
  - GPGPU using specialized units
- Exploiting regularity
  - Limitations of current GPUs
  - Dynamic data deduplication
  - Static data deduplication
- Conclusion

Knowing our baseline

- Design and run micro-benchmarks
  - Target NVIDIA Tesla architecture
  - Go far beyond published specifications
  - Understand design decisions
- Run power studies
  - Energy measurements on micro-benchmarks
  - Understand power constraints

S. Collange, D. Defour, A. Tisserand. Power consumption of GPUs from a software perspective. *ICCS* 2009.
S. Collange. Analyse de l'architecture GPU Tesla. *Technical Report hal-00443875*, Jan 2010.

Barra

- Functional instruction set simulator
- Modeled after NVIDIA Tesla GPUs
  - Executes native CUDA binaries
  - Reproduces SIMT execution
- Built within the Unisim framework
  - Unisim: ~60k shared lines of code
  - Barra: ~30k LOC
- Fast, accurate
- Produces low-level statistics
- Allows experimenting with architecture changes

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

S. Collange, M. Daumas, D. Defour, D. Parello. Barra: a parallel functional simulator for GPGPU. *IEEE MASCOTS*, 2010.

Primary constraint: power

- Power measurements on NVIDIA GT200

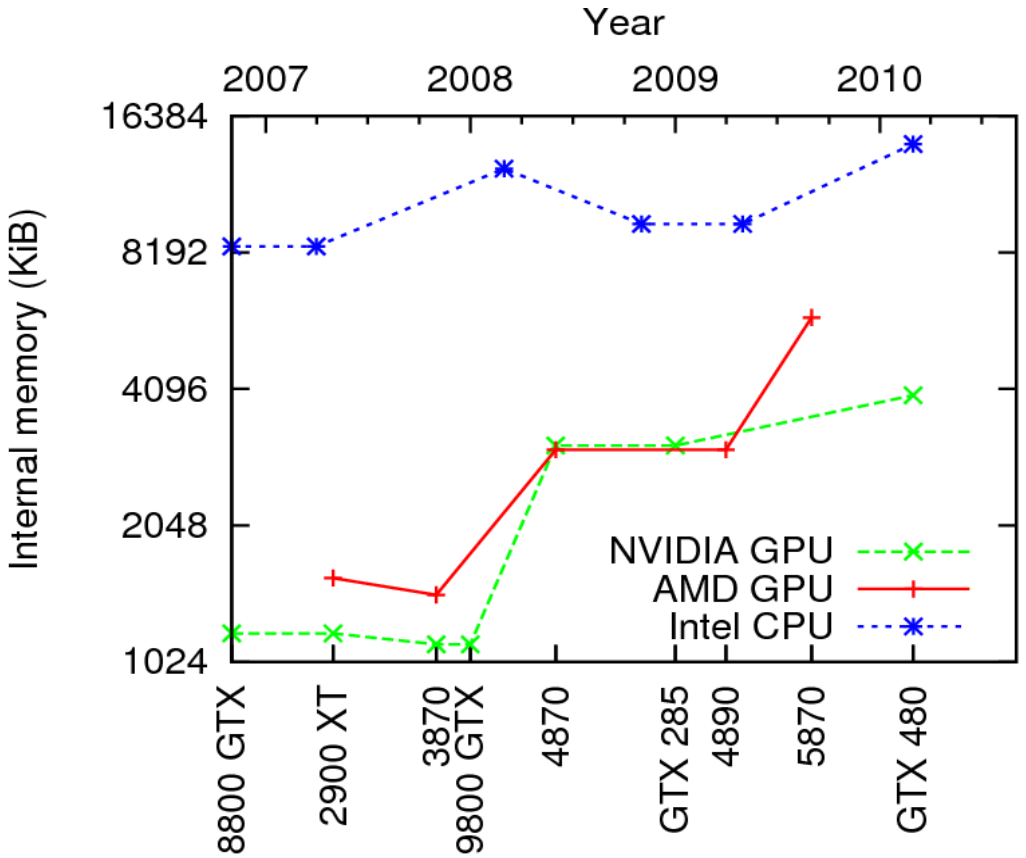
	Energy/op (nJ)	Total power (W)
Instruction control	1.8	18
Multiply-add on 32-element vector	3.6	36
Load 128B from DRAM	80	90

- With the same amount of energy
  - Read 1 word from DRAM
  - Compute 44 flops
- Need to keep memory traffic low
- Standard solution: caches

On-chip memory

- Conventional wisdom
  - CPUs have huge amounts of cache
  - GPUs have almost none
- Actual data

GPU	Register files + caches
NVIDIA GF100	3.9 MB
AMD Cypress	5.8 MB



- At this rate, will catch up with CPUs by 2012…

The cost of SIMT: register wastage

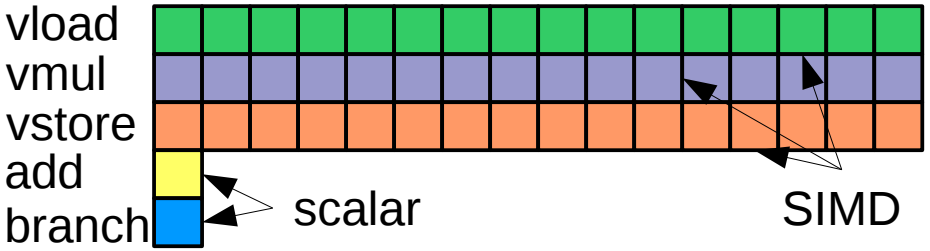
SIMD

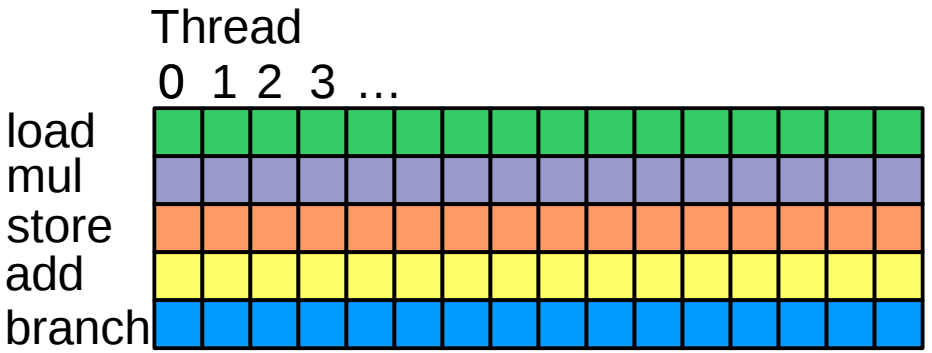
```
    mov     i ← 0
loop:
    vload   T ← X[i]
    vmul    T ← a×T
    vstore X[i] ← T
    add     i ← i+16
    branch i<n? loop
```

SIMT

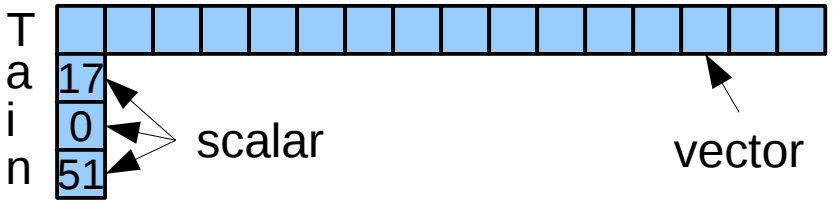
```
    mov    i ← tid
loop:
    load   t ← X[i]
    mul    t ← a×t
    store X[i] ← t
    add    i ← i+tnum
    branch i<n? loop
```

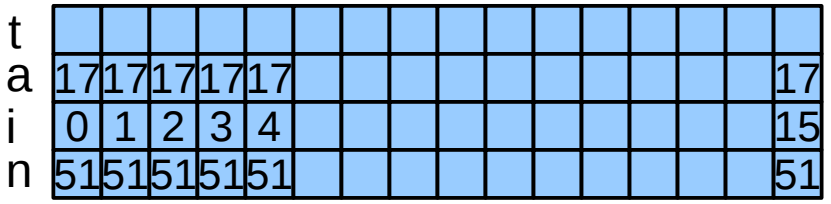
- Instructions





- Registers



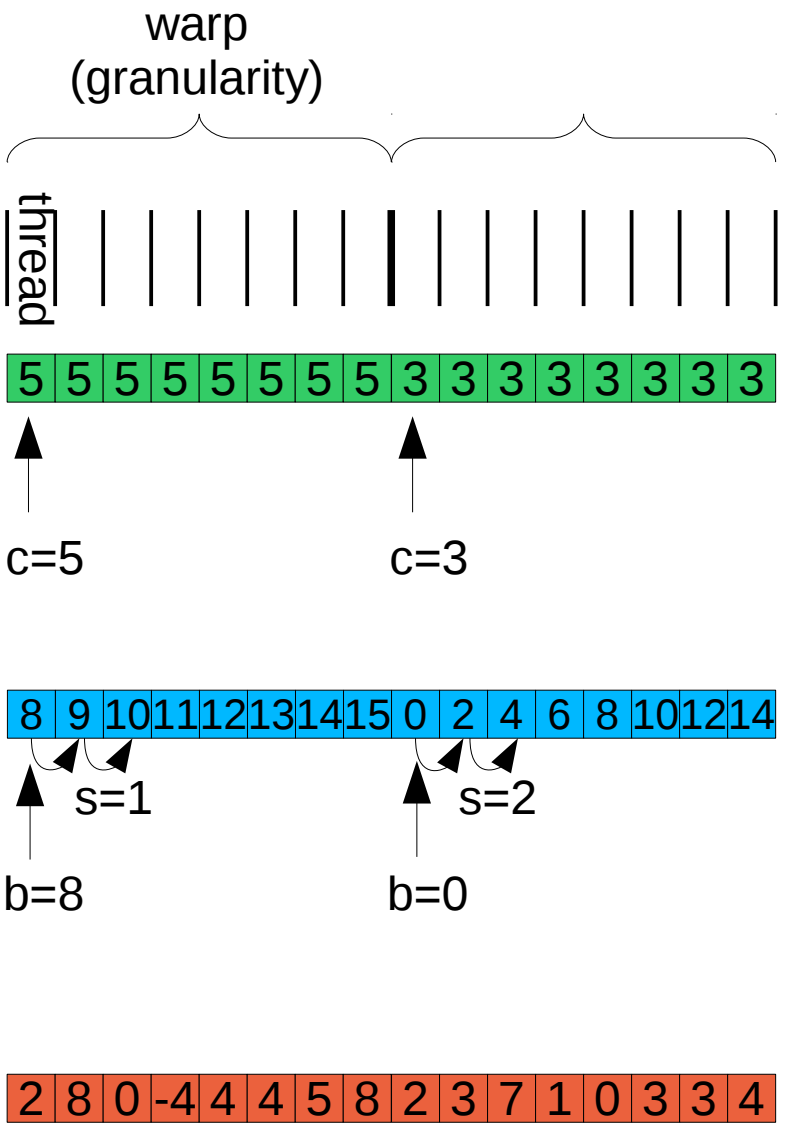


SIMD vs. SIMT

	SIMD	SIMT
Instruction regularity	Vectorization at compile-time	Vectorization at runtime
Control regularity	Software-managed Bit-masking, predication	Hardware-managed Stack, counters, multiple PCs
Memory regularity	Compiler selects: vector load-store or gather-scatter	Hardware-managed Gather-scatter with hardware coalescing
Data regularity	Scalar registers, scalar instructions	*Duplicated* registers, *duplicated* ops

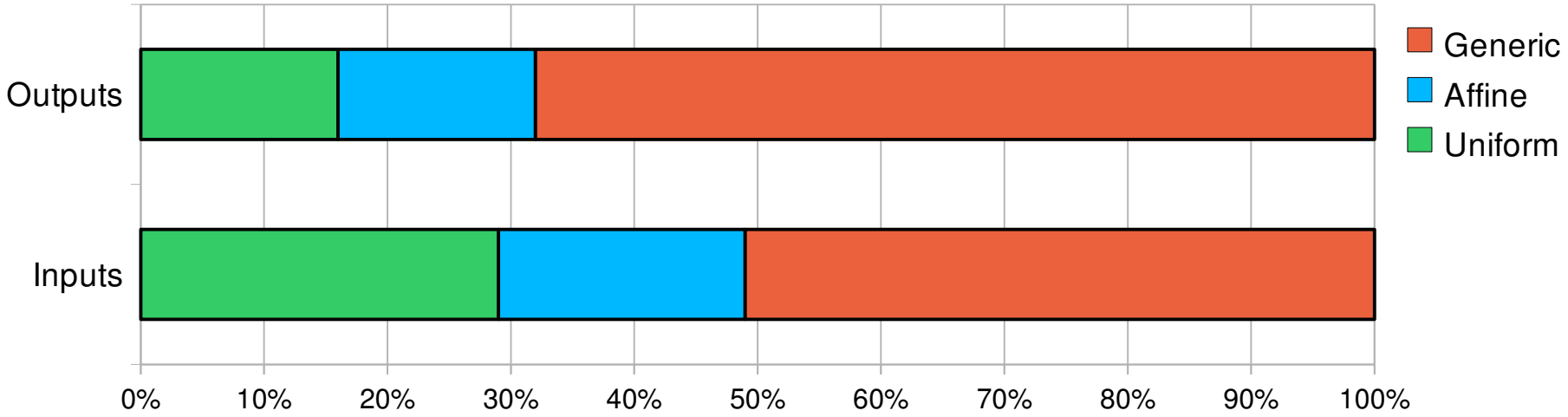
Uniform and affine vectors

- Uniform vector
  - In a warp, v[*i*] = *c*
  - Value does not depend on lane ID
- Affine vector
  - In a warp, v[*i*] = *b* + *i 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



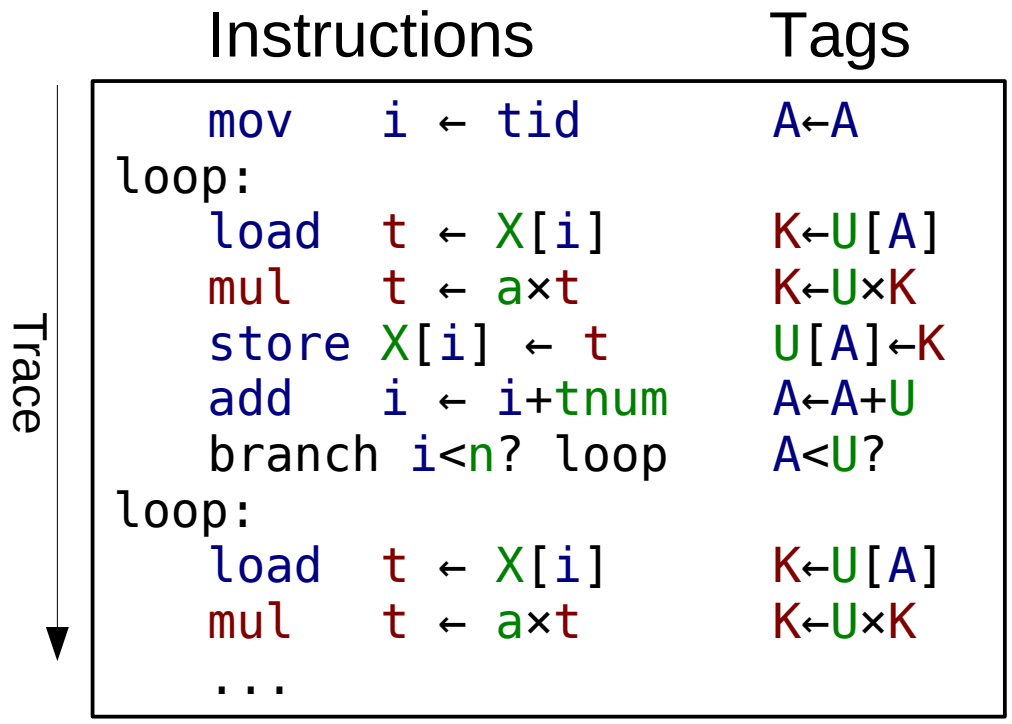
  - 49% of all reads from the register file are affine
  - 32% of all instructions compute on affine vectors
- This is what we have “in the wild”
  - How to capture it?

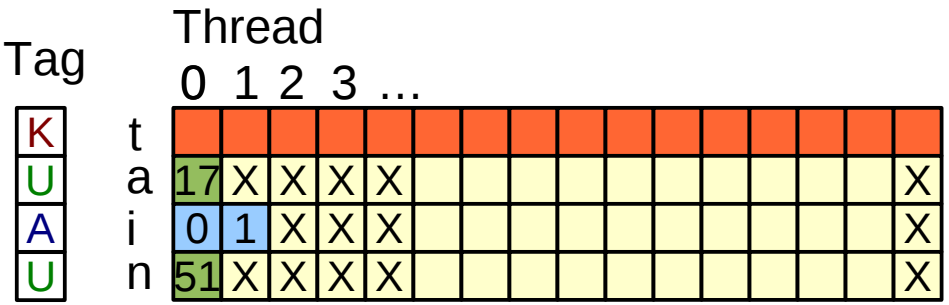
Outline of this talk

- Introduction to GPU architecture
- Balancing specialization and genericity
  - Current challenges
  - GPGPU using specialized units
- Exploiting regularity
  - Limitations of current GPUs
  - Dynamic data deduplication
  - Static data deduplication
- Conclusion

Tagging registers

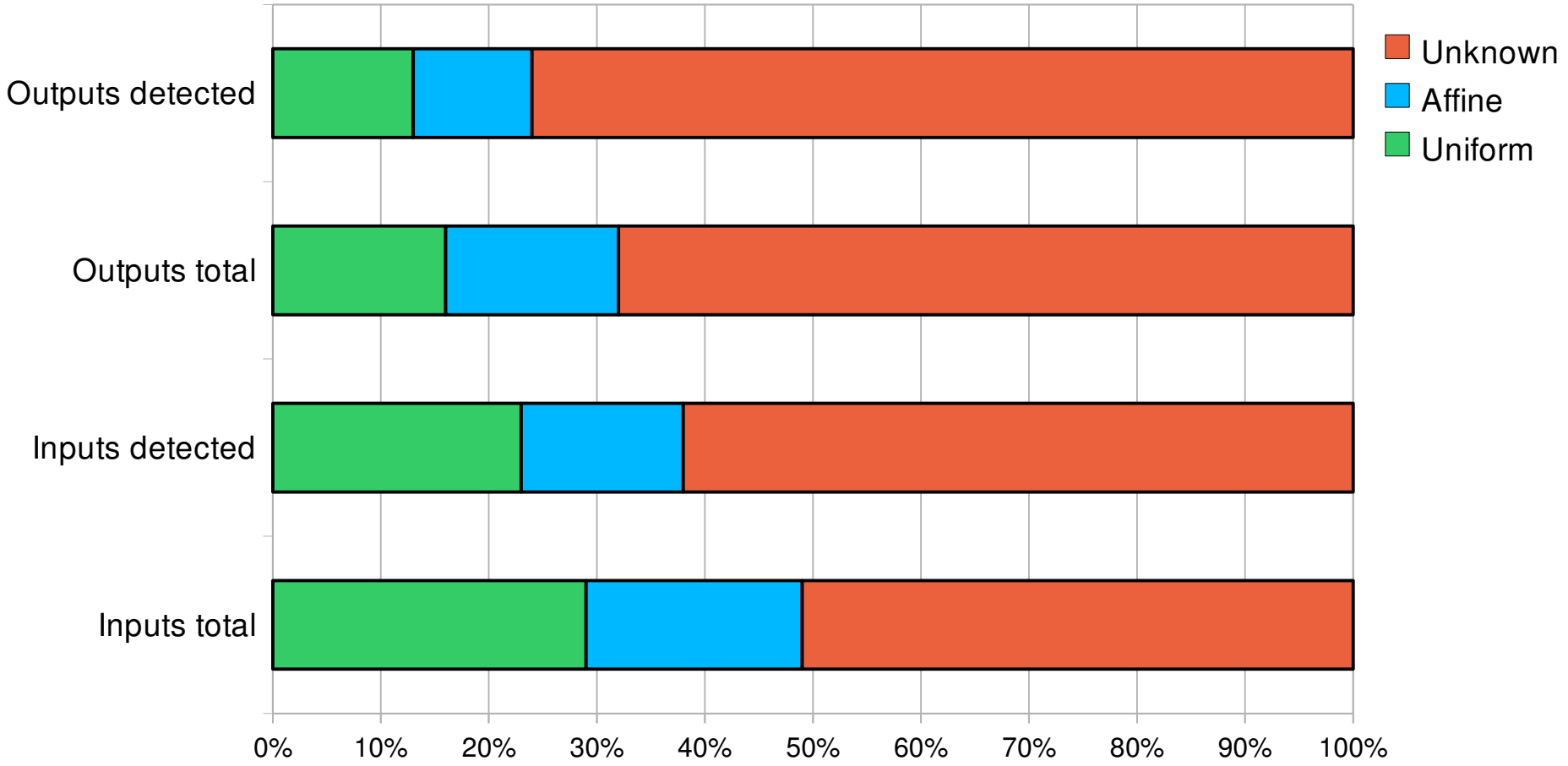
- Associate a tag to each vector register
  - **U**niform, **A**ffine, un**K**nown
- Propagate tags across arithmetic instructions
- 2 lanes are enough to encode uniform and affine vectors





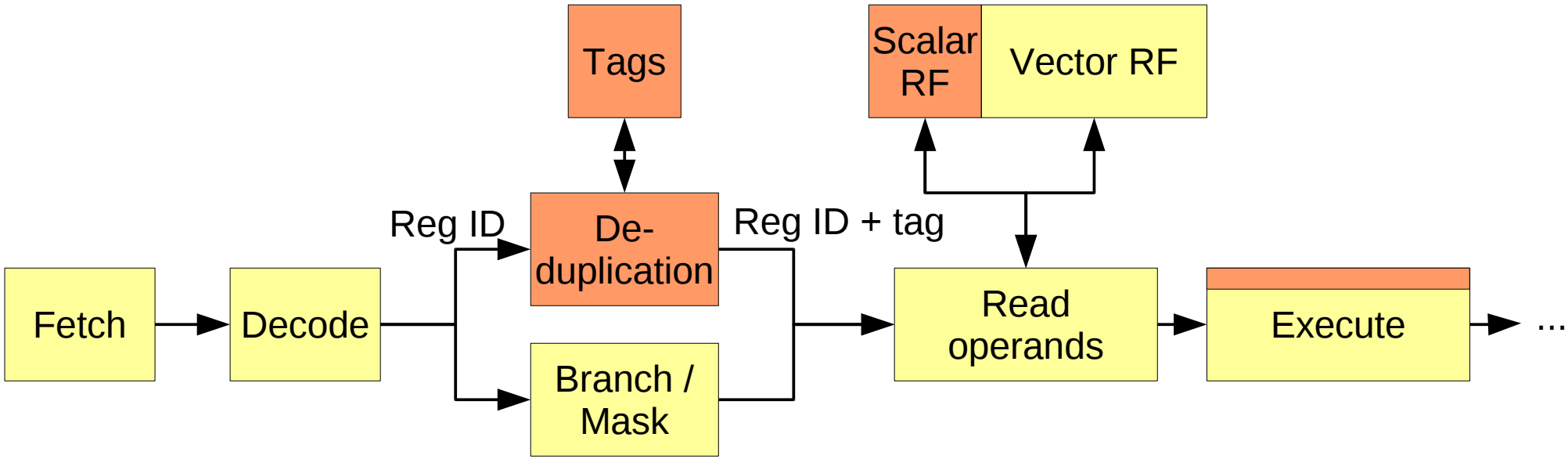
Dynamic data deduplication: results

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

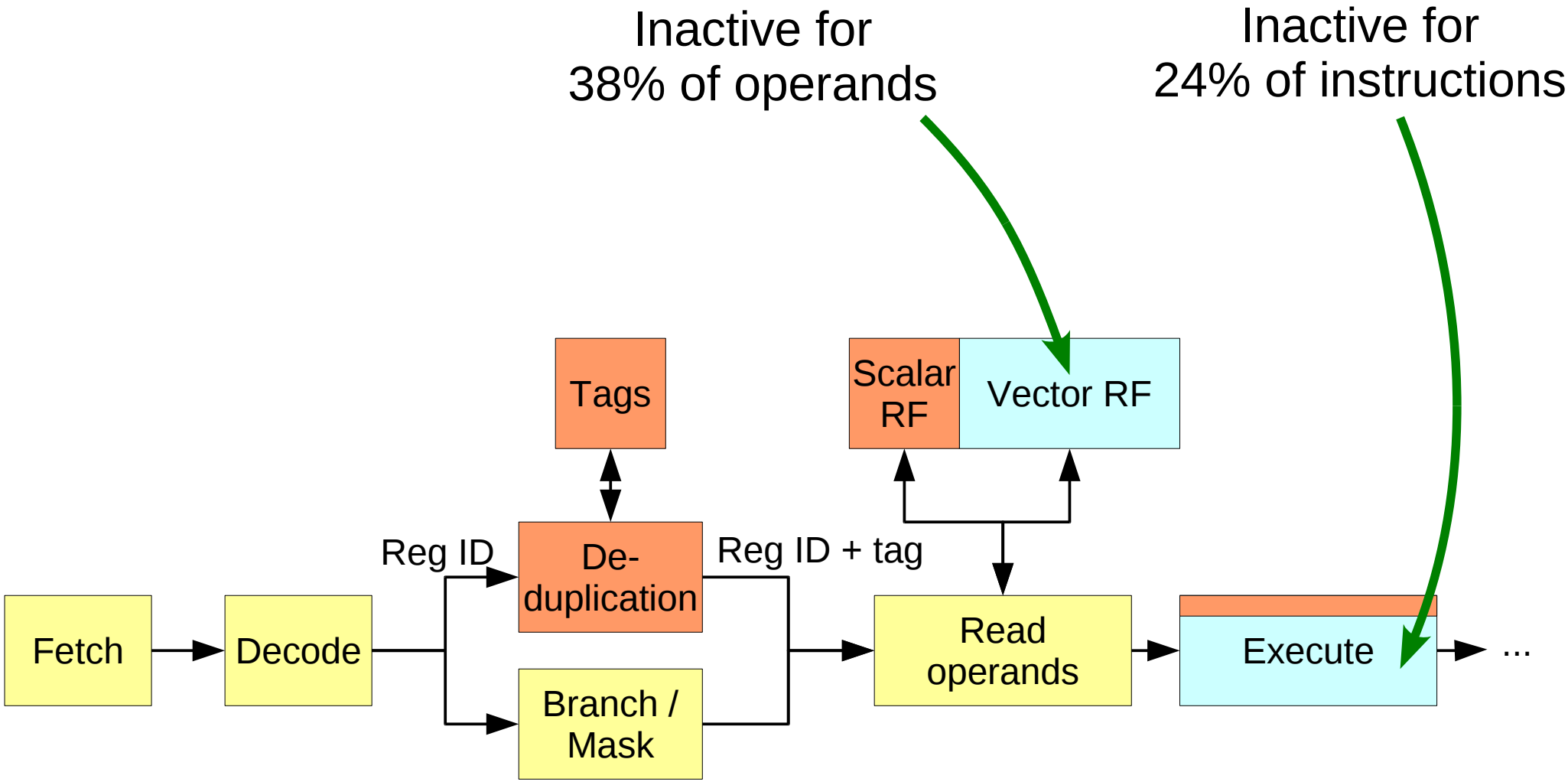


New pipeline

- 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



Power savings



S. Collange, D. Defour, Y. Zhang. Dynamic detection of uniform and affine vectors in GPGPU computations. Europar HPPC09, 2009

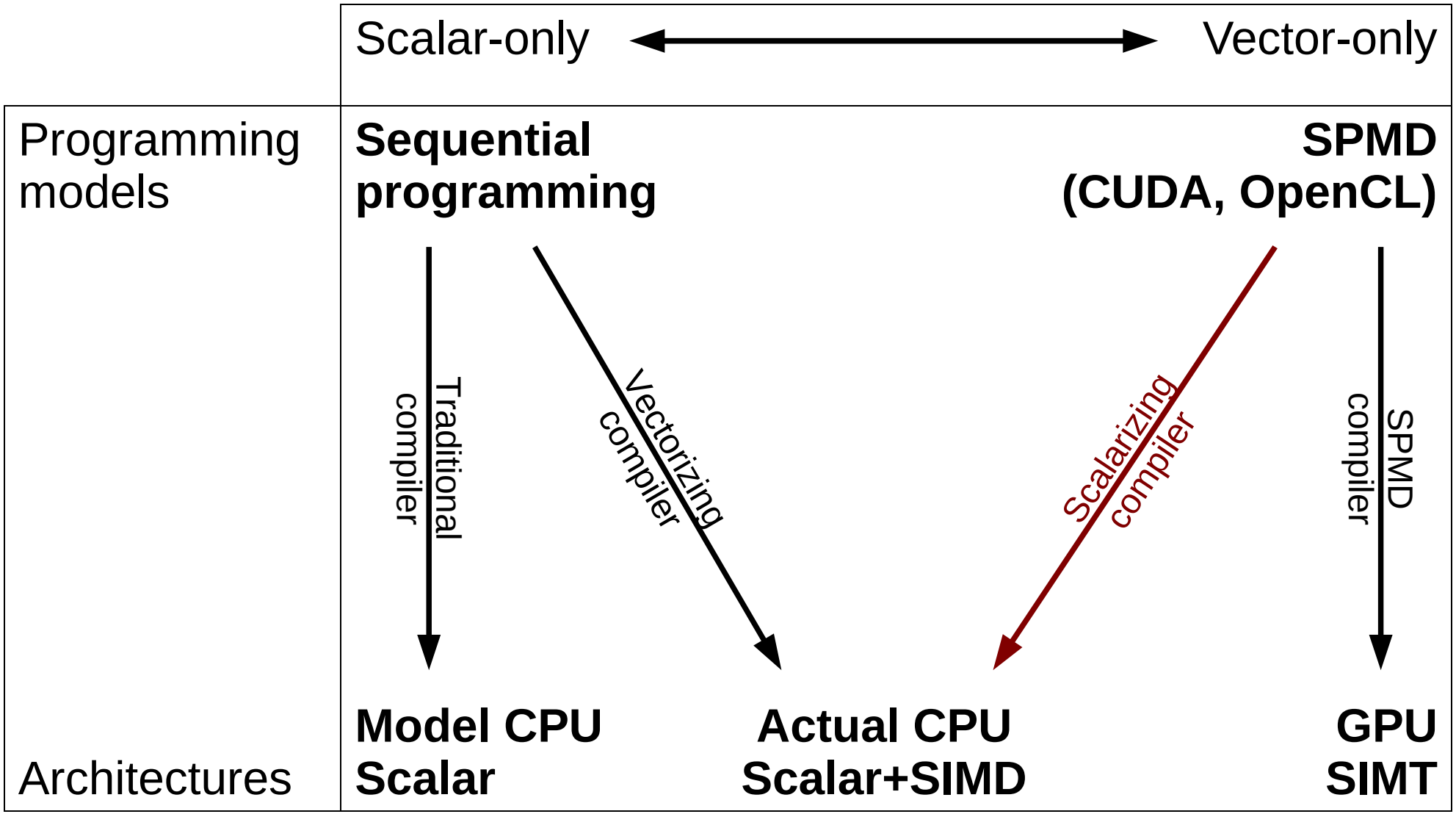
SIMD vs. SIMT

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

Outline of this talk

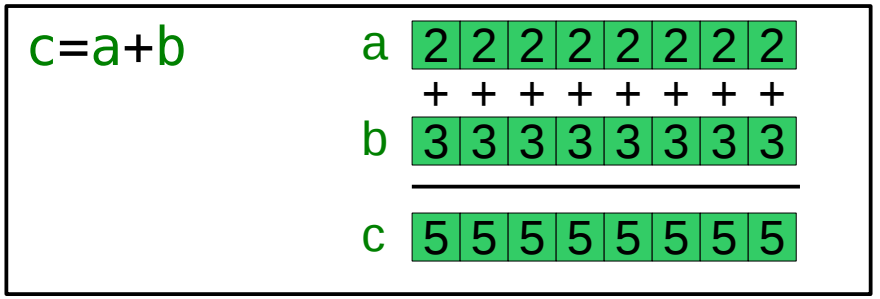
- Introduction to GPU architecture
- Balancing specialization and genericity
  - Current challenges
  - GPGPU using specialized units
- Exploiting regularity
  - Limitations of current GPUs
  - Dynamic data deduplication
  - Static data deduplication
- Conclusion

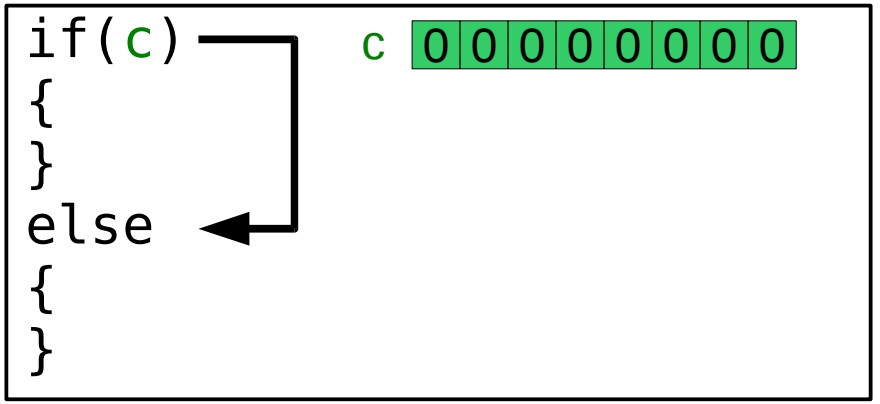
A scalarizing compiler?

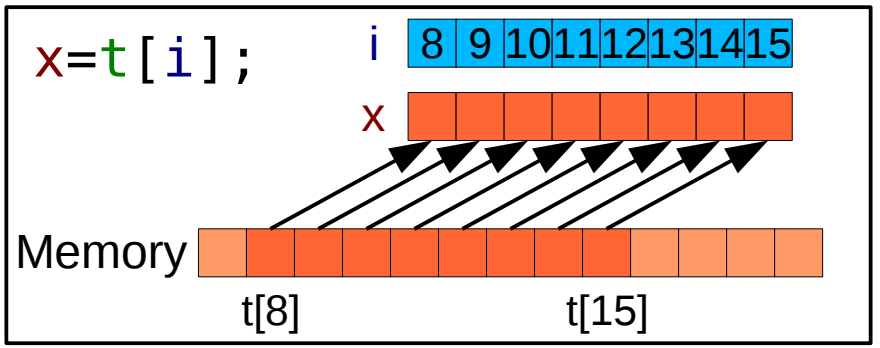


Still uniform and affine vectors

- 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



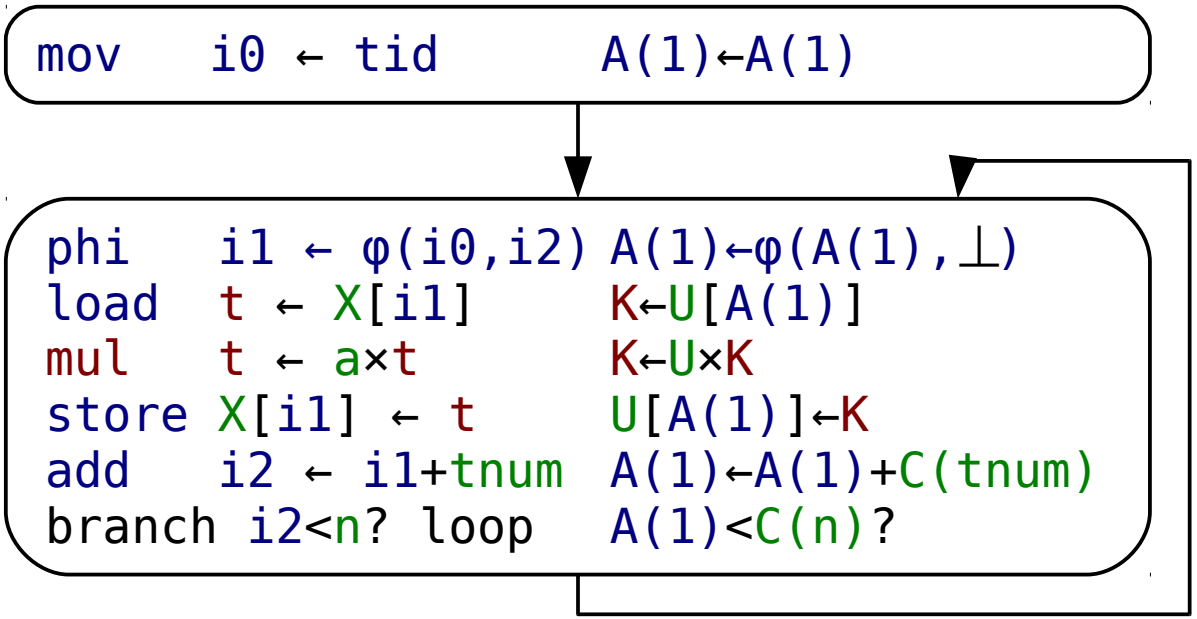




From SPMD to SIMD

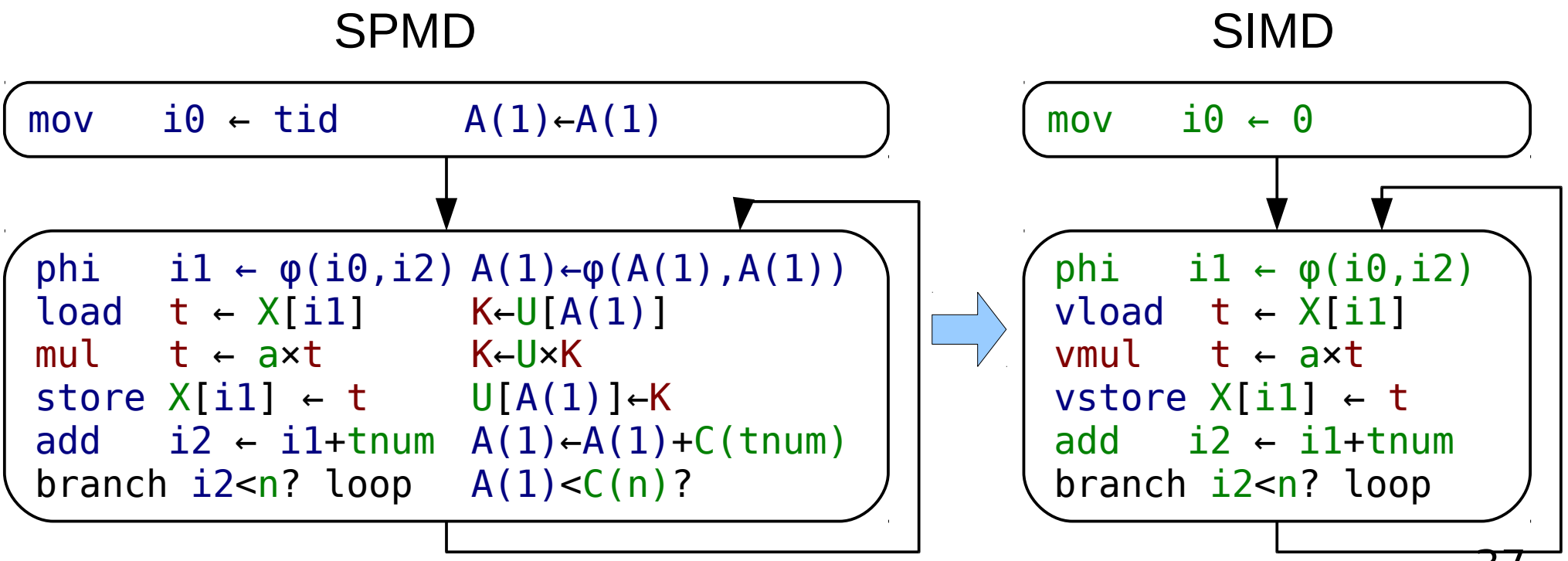
- Forward dataflow analysis
- Statically propagate tags in dataflow graph
  - ⊥, C(v), U, A(s), K
  - Propagate value *v* of constants, stride *s* of affine vectors, and alignment

SPMD



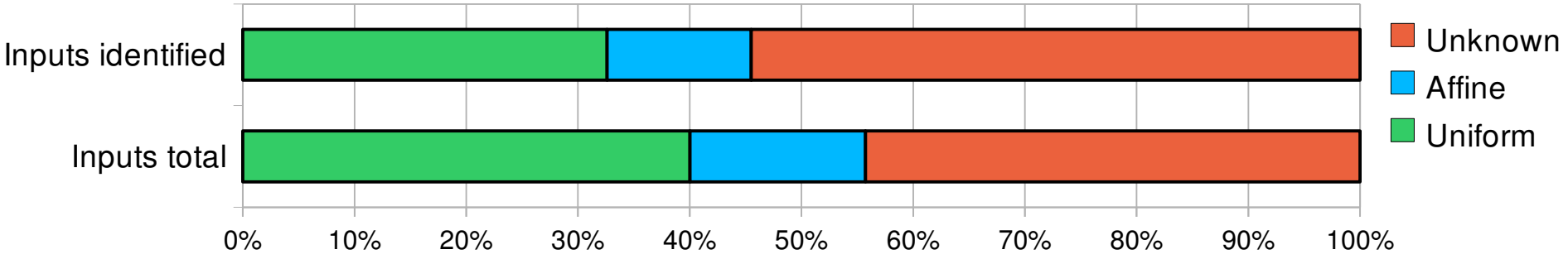
From SPMD to SIMD

- 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

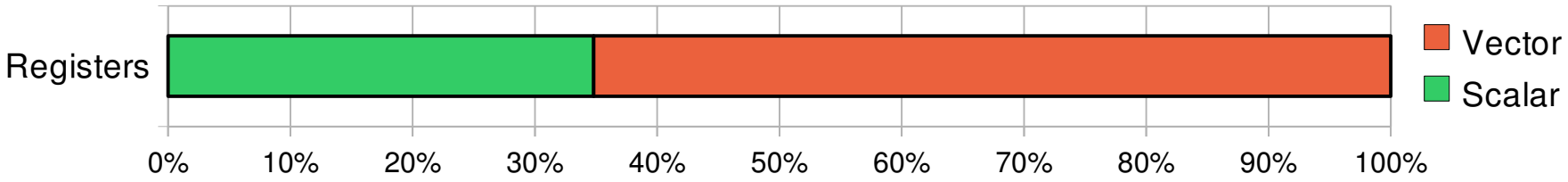


Results: instructions, registers

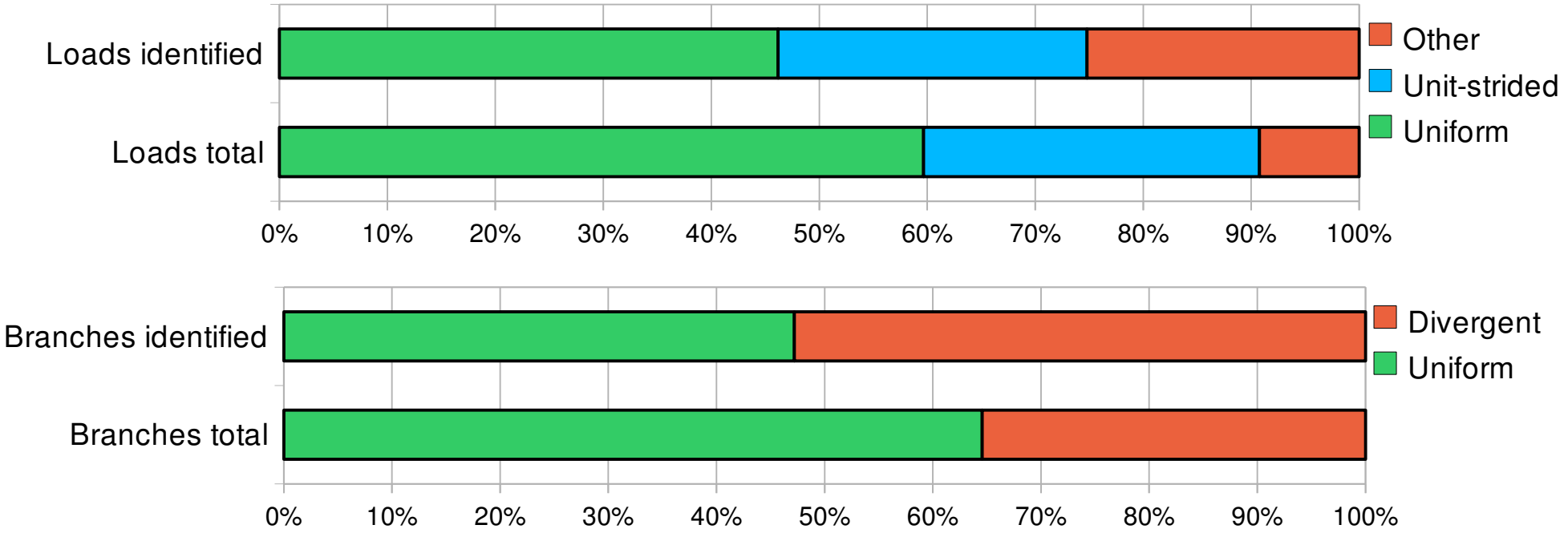
- Benchmarks: CUDA SDK, SGEMM, Rodinia, Parboil
- Static operands



- Split register allocation



Results: memory, control



- 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

Static deduplication	Dynamic deduplication
Allows simpler hardware	Preserves binary compatibility
Governs register allocation and instruction scheduling	Captures dynamic behavior
Enables constant propagation	Unaffected by (future) software complexity
Applies to memory (call stack…)	

Summary of contributions

- Specialized units can be used for other applications
  - Make specialized units (more) configurable at reasonable cost
- 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

Perspectives

- New computer architecture field:
  SIMT microarchitecture
  - Improve flexibility on irregular applications
  - Improve efficiency on regular applications
- Can we bridge the SIMD – SIMT gap?
  - Hints from compiler, keep microarchitecture freedom
- Short-term: extend redundancy elimination to the memory hierarchy
  - Data compression in caches, memory
  - Consider floating-point data