

Coherent Threading

Coherent Vector Lane Threading
(SIMT, DIMT, NIMT)

Microarchitectures Intermediate
Between SISD and MIMD
Scalar and SIMD parallel vector

Disclaimer

The content of this presentation is my personal opinion only.

The statements I make here in no way represent my employer's position, nor am I authorized to speak on behalf of my employer.

This is not a product announcement, nor an announcement of product details. Nor even an announcement of a research direction

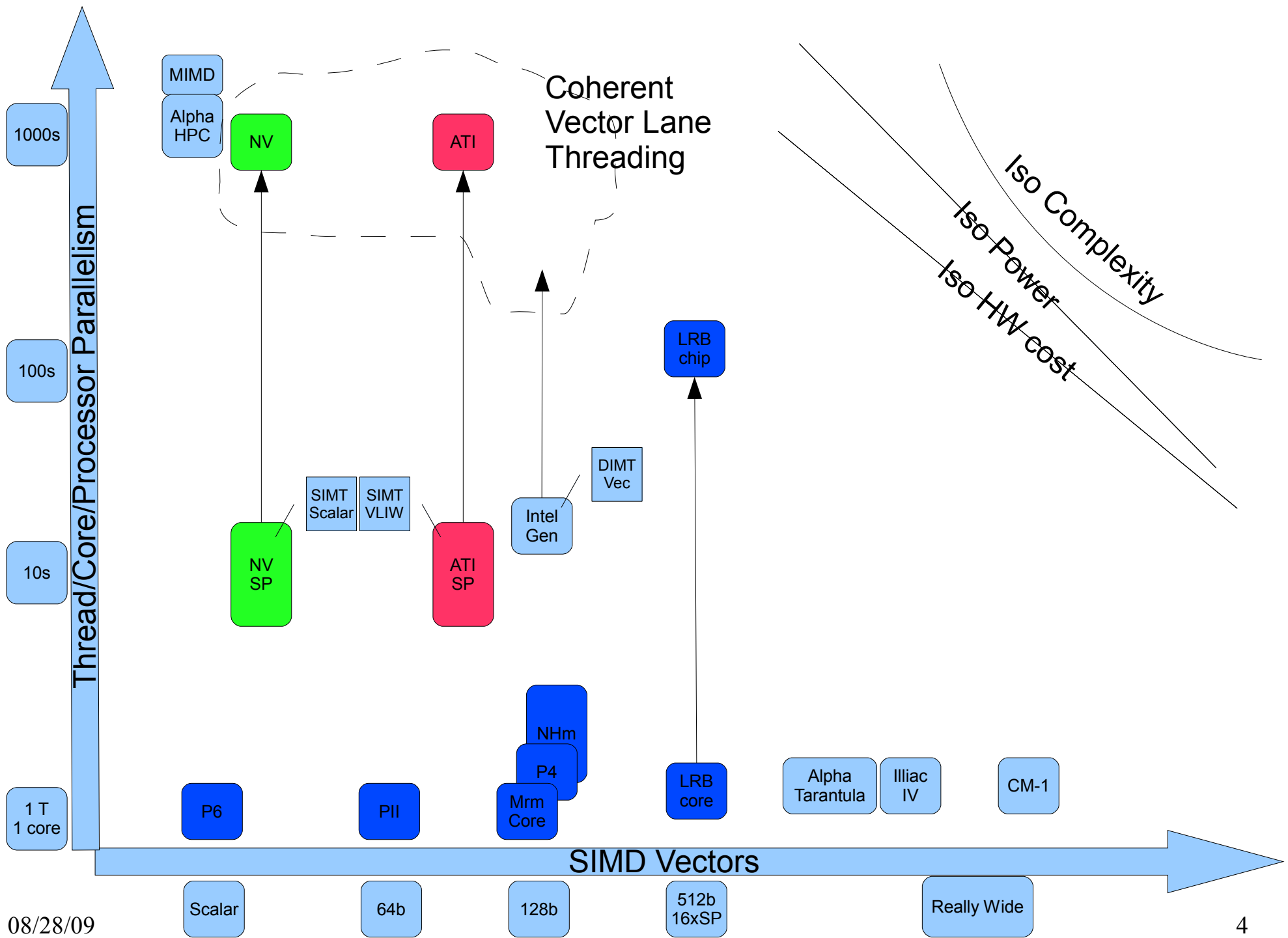
Although I am an employee - currently of Intel, in the past of other computer companies such as AMD, Motorola, and Gould - I reveal this only so that the reader may account for any possible bias I may have towards my employer's products.

In fact, this posting may not even represent my personal opinion, since occasionally I play devil's advocate.

Terminology

- Time pipelined vectors = e.g. Cray1
- SIMD parallel vectors = e.g. Convex, e.g. DEC Tarantula, e.g. LRB
- Thread = standard software term
= PC + SP + regs. Multiple threads in same virtual address space.
- Intra-lane or Cross Lane Latency Tolerant, Threads
 - Commonly called Fine Grain Multithreading; IMHO a horribly misleading term, e.g. Burton Smith HEP threads
 - e.g. SMT, Wmt / Nhm HT, LRB threads,
 - SoEMT, Interleaved MT, timesliced threads
- Vector Lane Threading = Glewism... Coherent Vector Lane Threading Coherent Threading...
= Kozyrakis (Berkeley) ?
 - Unfortunately, their VLT different than Glew use of VLT
 - SIMT = Single Instruction Multiple Threads = Nvidia terminology
 - DIMIT = Dual Instruction Multiple Threads = e.g. Intel Gen?
 - NIMT = N instructions M threads = Glewism
- Coherent Parallel Thread Grouping
 - Nvidia “warp” (of threads)
 - Intel GenX “thread” (of channels)
 - ATI “wavefront”
- Vector lane thread
 - Nvidia “thread”
 - Intel GenX “channel”
- Lane – aligned element of SIMD vector datapath
 - “Physical Lane” - as above, actually implemented
 - “Virtual Lane” - new Glewism, stuff associated with aligned elements of a SIMD vector datapath, possibly time multiplexed over multiple clock cycles
 - e.g. Asanovic's Virtual Processors
- Coherency – GPU term, similarity between threads.
NOT cache or memory consistency/coherency.

Terminology in this presentation NOT consistent,
reflects evolution/accumulation.



GPU Shader Engines
are NOT
Your Family's Vector Processor

Neither Uncle Seymour (Cray)

Vector Time Pipelined

Nor Uncle Steve (Wallach)

Vector Parallel

Andy Glew

or...

How to use GPU-like
Coherent Vector Lane Threading
(aka SIMT, aka DIMT, aka NIMT)
to increase vector ALU efficiency
over SIMD vector parallel
microarchitectures

Agenda

- Introduction
 - The Point of this Presentation
 - Scalar, MIMD, SIMD
 - MCMT (+ optimizations – SIMT)
 - SIMT (classic stack), DIMT, NIMT, Coherent Vector Lane Threading
- Inefficiencies in SIMD parallel vector processors
 - VL / Vmask / Lane crossing
 - Time pipelined vectors ... VL ... SIMT ifetch
 - Fundamental SIMT advantage > predication
 - Loop buffer: SIMD → SIMT → MIMD
 - Rejiggering Threads in Warp
 - More rebalancing
- Conclusions
 - Summary of SIMT VPU utilization improvements
 - Top 10 reasons why SIMT CVLT better than SIMD
- *Backup*
 - *To learn more*
 - *VLIW, MCMT*
 - *SIMD on SIMT...*
 - *Data pattern power*
 - *SIMT register files*
 - *Speed, size*
 - *Variable #lregs*
 - *Wires*
 - *Rejiggering*
 - *Are SIMT threads latency tolerant threads?*
Are SIMT threads really threads at all?
 - *X86 on SIMT*
 - *SIMT ISA extensions?*
 - *Reliability, Lockstep*
 - *Instruction and Data Coherence*
Intra vs. Intra Cacheline Threading
 - *Why SIMT is friendlier to software*
 - *Strawman Microarchitecture*
 - *Terminology*

Modern GPUs use many Uarch Techniques

- Many “processor cores”
- Barrel processing
 - Burton Smith HEP style threads – hide latencies, avoid interlocks
- SoEMT threading

- **Vector Lane Threading within the cores**
 - **aka SIMD, SIMT, ...**
 - **PC per vector lane**
- Clustering within the “vector” width
 - e.g, $4 \times 32 = 128\text{b}$ clusters, for 32 wide SIMD

- Vector-time-pipelining
- Vector-width-parallelization
- ATI: VLIW within a vector lane

The Point of this Presentation

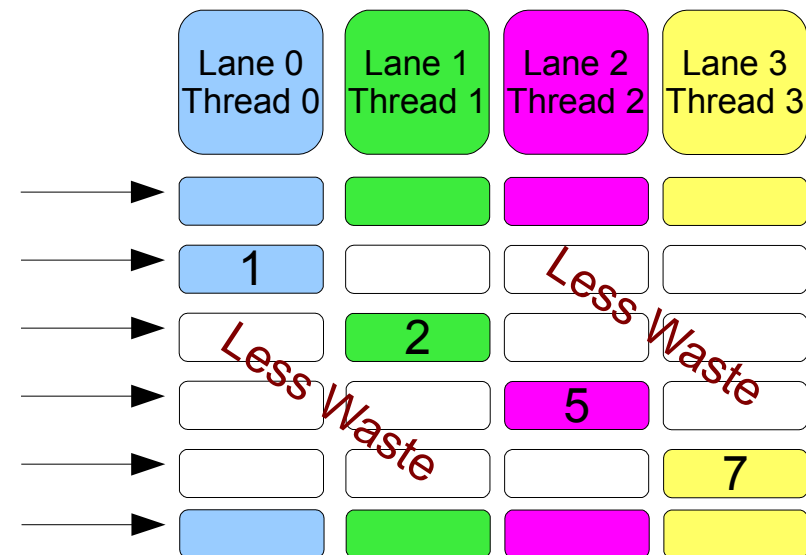
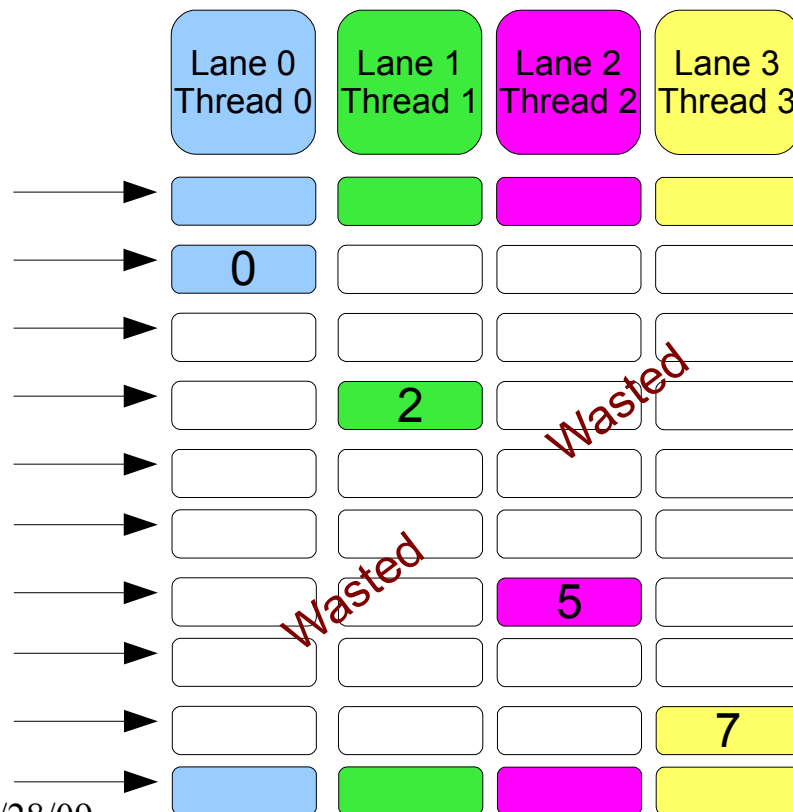
- Typical GPUs use
Coherent Vector Lane Threading, aka SIMT, aka SIMD
- Many people seem to believe this equivalent to
“conventional” vector processors
- NOT your Father's Vector Processor
 - SIMT is not just vectors with mask and good scatter/gather
 - Reduces key sources of inefficiency in Parallel Vector Processors
- Coherent Vector Lane Threading
is to Parallel Vector Processors
what Out-of-Order Execution is to VLIW
 - Microarchitecture, not Macroarchitecture
 - Same “ISA Concept” runs on scalar MIMD
 - Assuming you /encouraging you to have lots of threads

Quick Examples of Effects of SIMT/DIMT/NIMT Coherent Vector Lane Threading Optimizations on VPU Utilization

Just look at the pictures.
We'll see them again later, with explanation.

Predication executes all paths; SIMT only executes instructions used at least once

```
switch(f(i)) {  
  case 0: I0; break;  
  case 1: I1; break;  
  ...  
  case 7: I7; break;  
}
```



Why not a loop buffer at each lane?

```

I0
for all threads l
  repeat 100
    for all vector elements j
      l1: if cond[i,j] then
        l2: v[j] = ...

```

I3

Where $\text{cond}[i,j] =$
arbitrary, e.g.

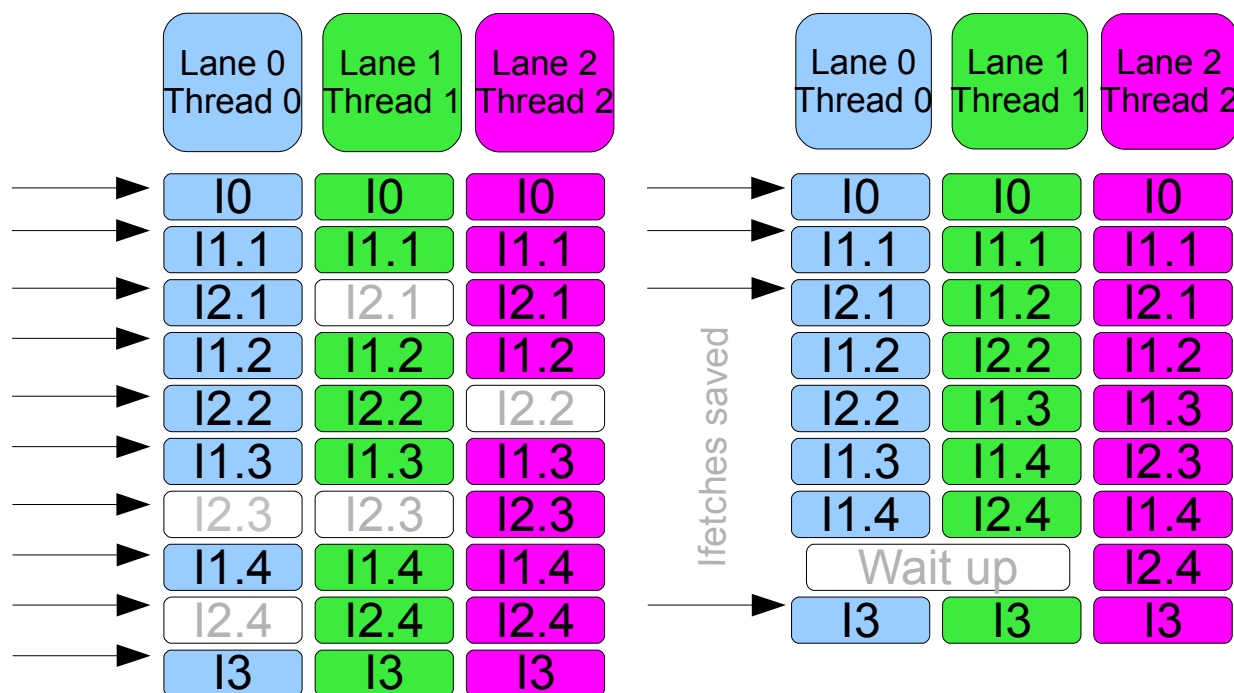
101

110

000

011

Assuming $\text{cond}[i,j]$
is NOT available to
instruction sequencer



Rejiggering Threads between Cycles of a multi-cycle Warp

Original, Imbalanced = 50% utilized



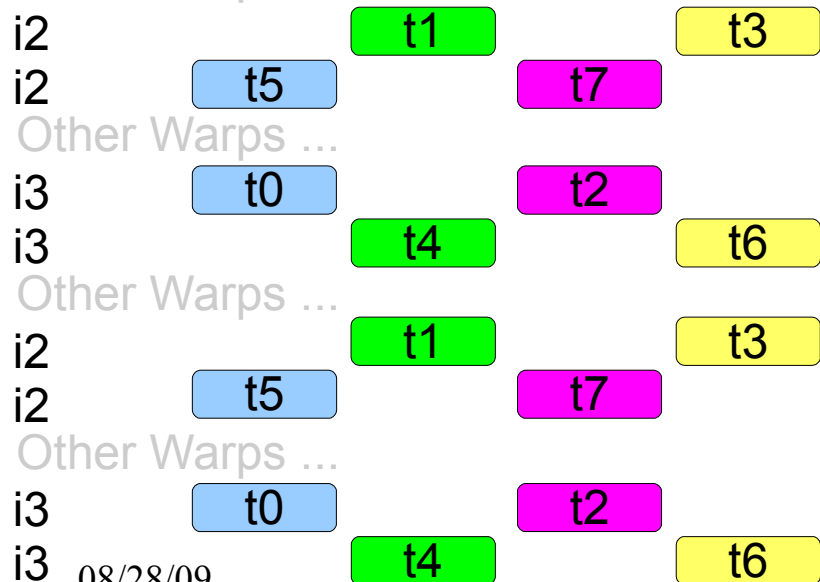
Example assumes
Thread skewing
between waves
in a wavefront

```

I1
repeat i=0 to ...
  for threads t=0 to 7
    if even(i) XOR even (t)
      then I2
    else I3
I4
    
```

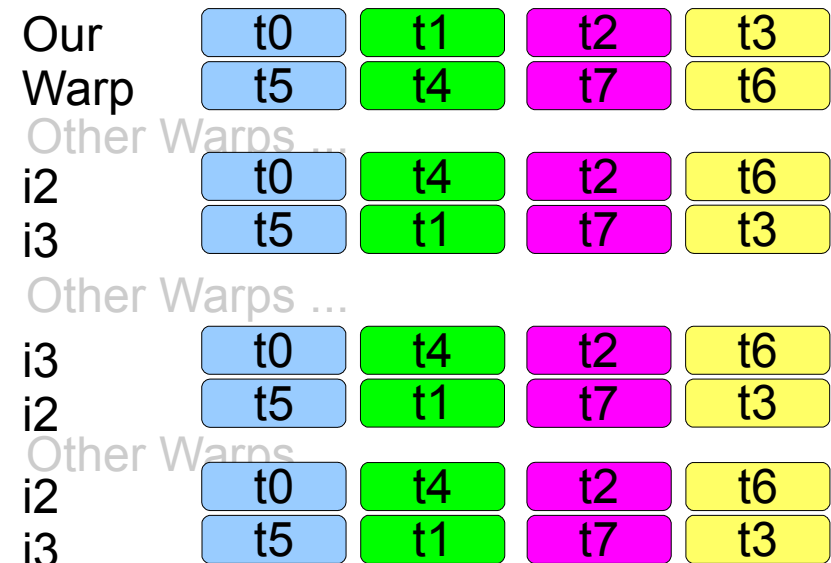
Our
Warp

Other Warps ...



08/28/09

Rejiggered within Wavefront = 100% utilized



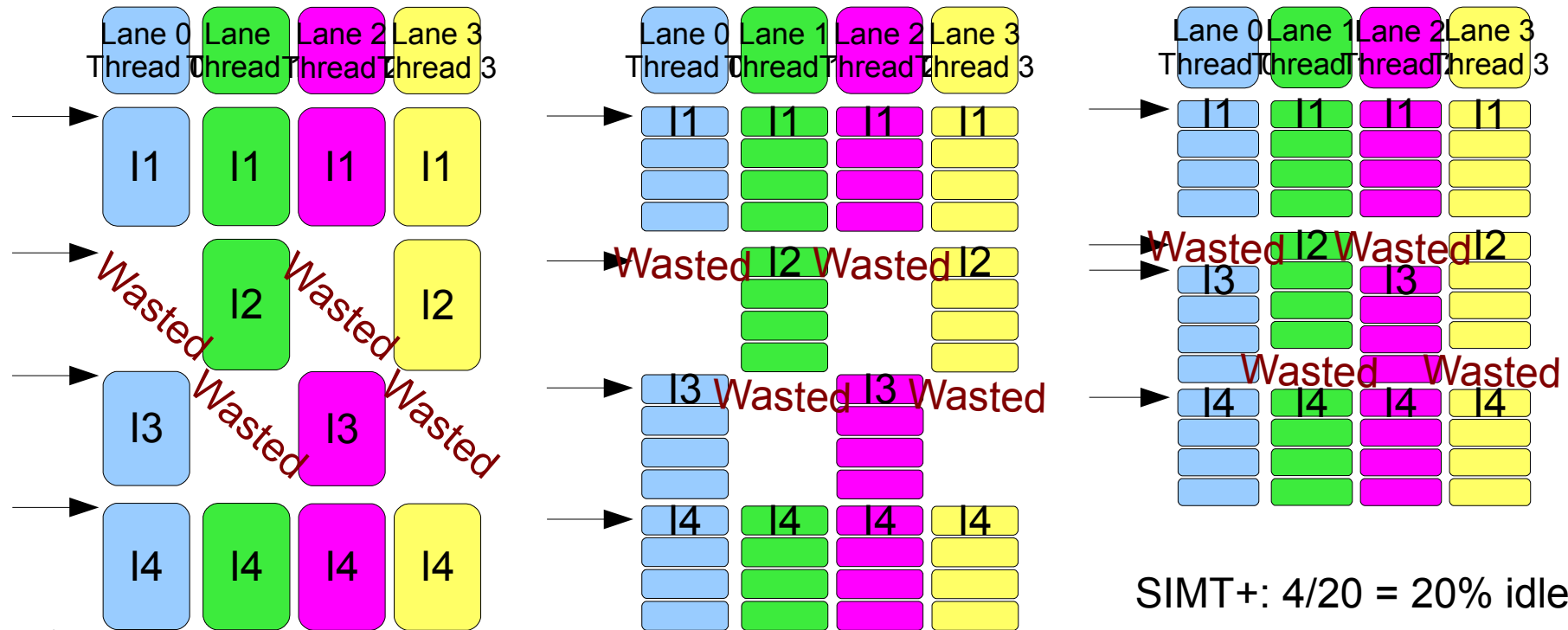
Other Warps ...

```

l1
for all i
  if odd
    then l2
  else l3
  endif
end loop
l4

```

SIMT dispatch during idle ifetch while Time Pipelined Vectors are busy



SIMT+: $4/20 = 20\%$ idle

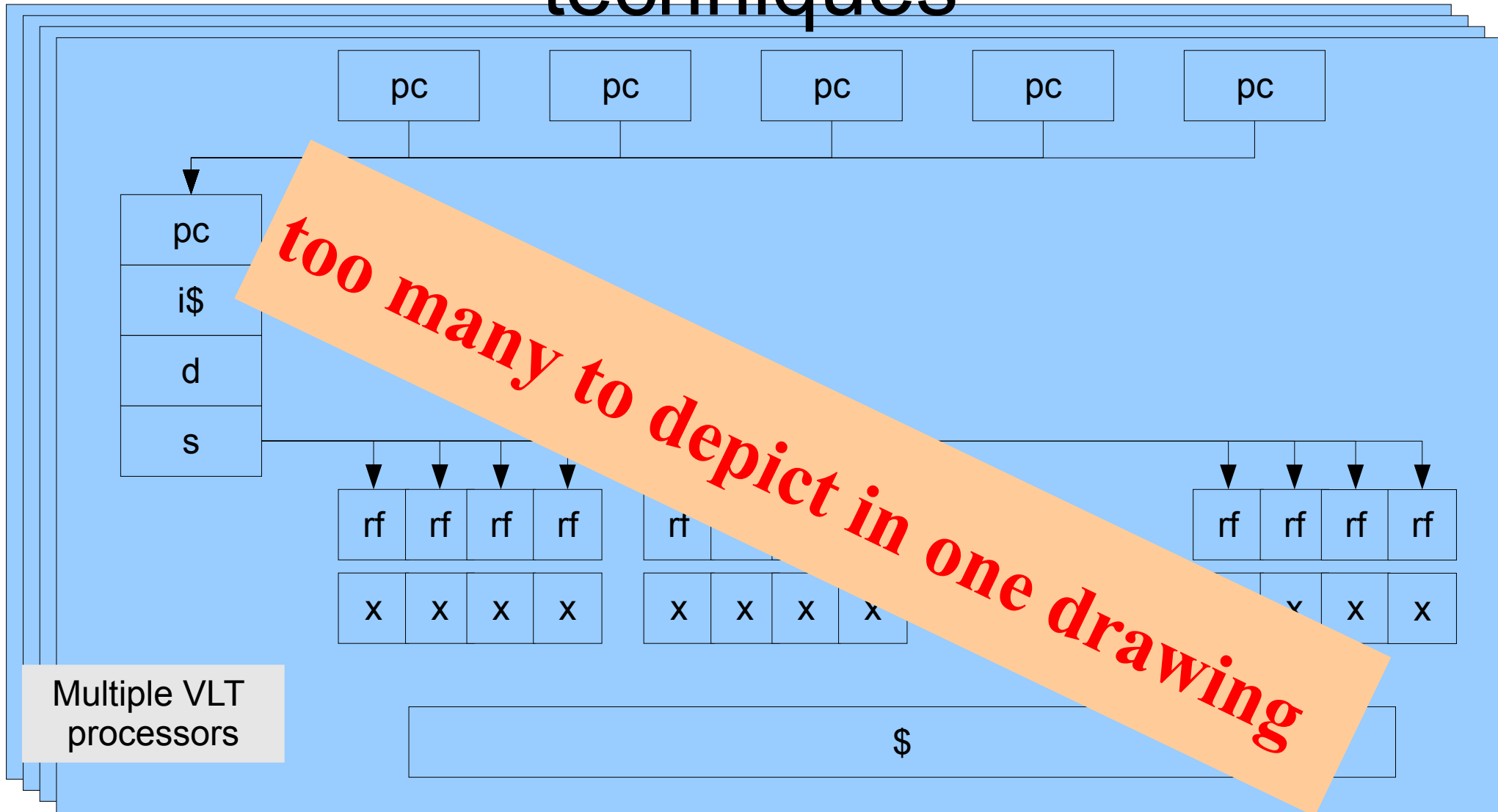
37.5% speedup

End of Examples.

Back to Explanations.

We'll see the pictures later.

Modern GPUs use many Varch techniques



Modern GPUs use many Uarch techniques

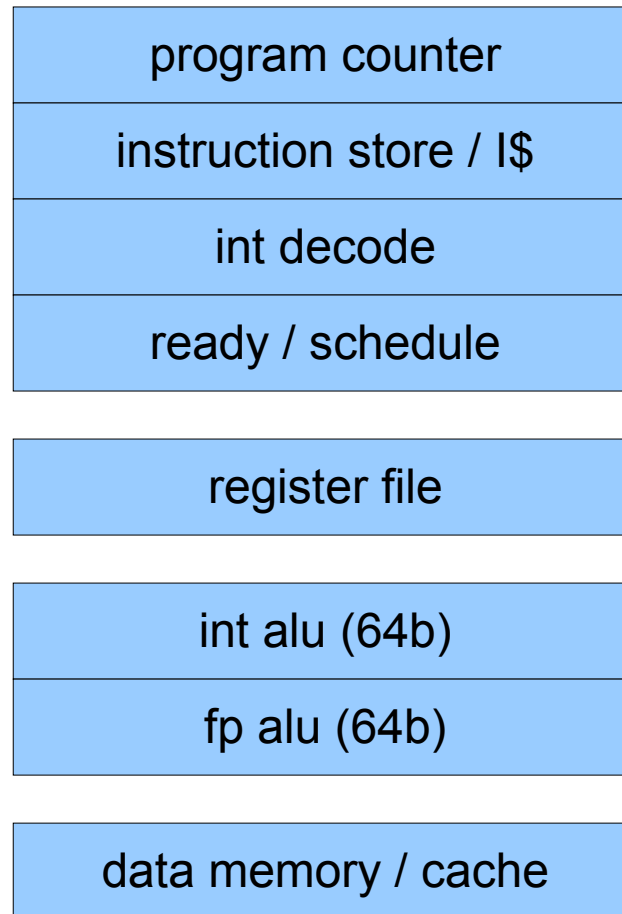
- Fixed Function Logic
- Many processor cores
- Multiple threads per core (latency tolerance)
- Superpipelining
- SIMD vector parallelism
- Vector lane threading (aka SIMD)
- Clusters of execution units
- Variable # registers per thread (to support variable size register file)
- Scatter gather and strided access in the register file
- Scatter gather and strided access to memory
- Caches ... or not

too many to depict in one slide

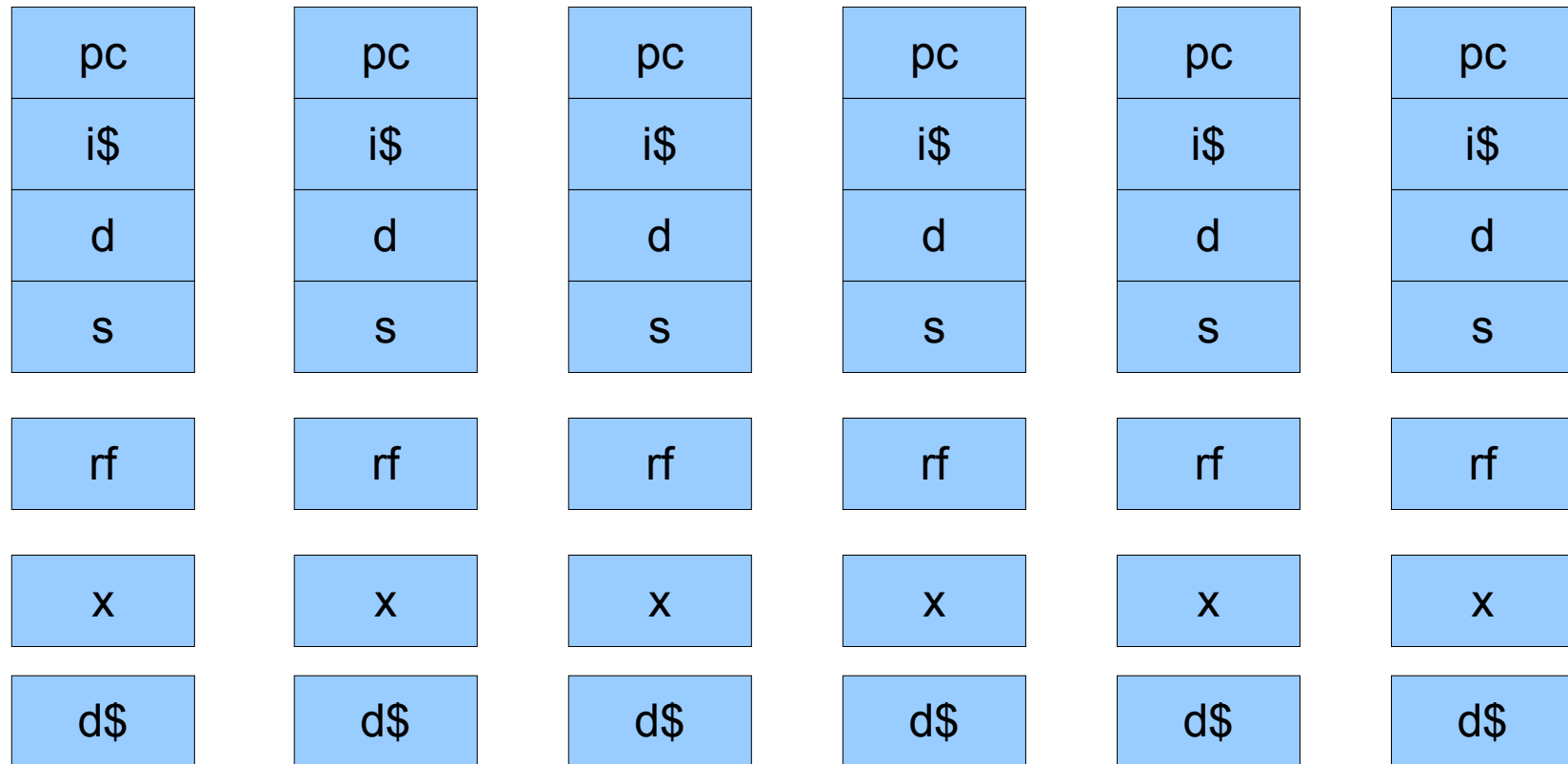
Modern GPUs use many Uarch techniques

So let's look at some uarch techniques in isolation...

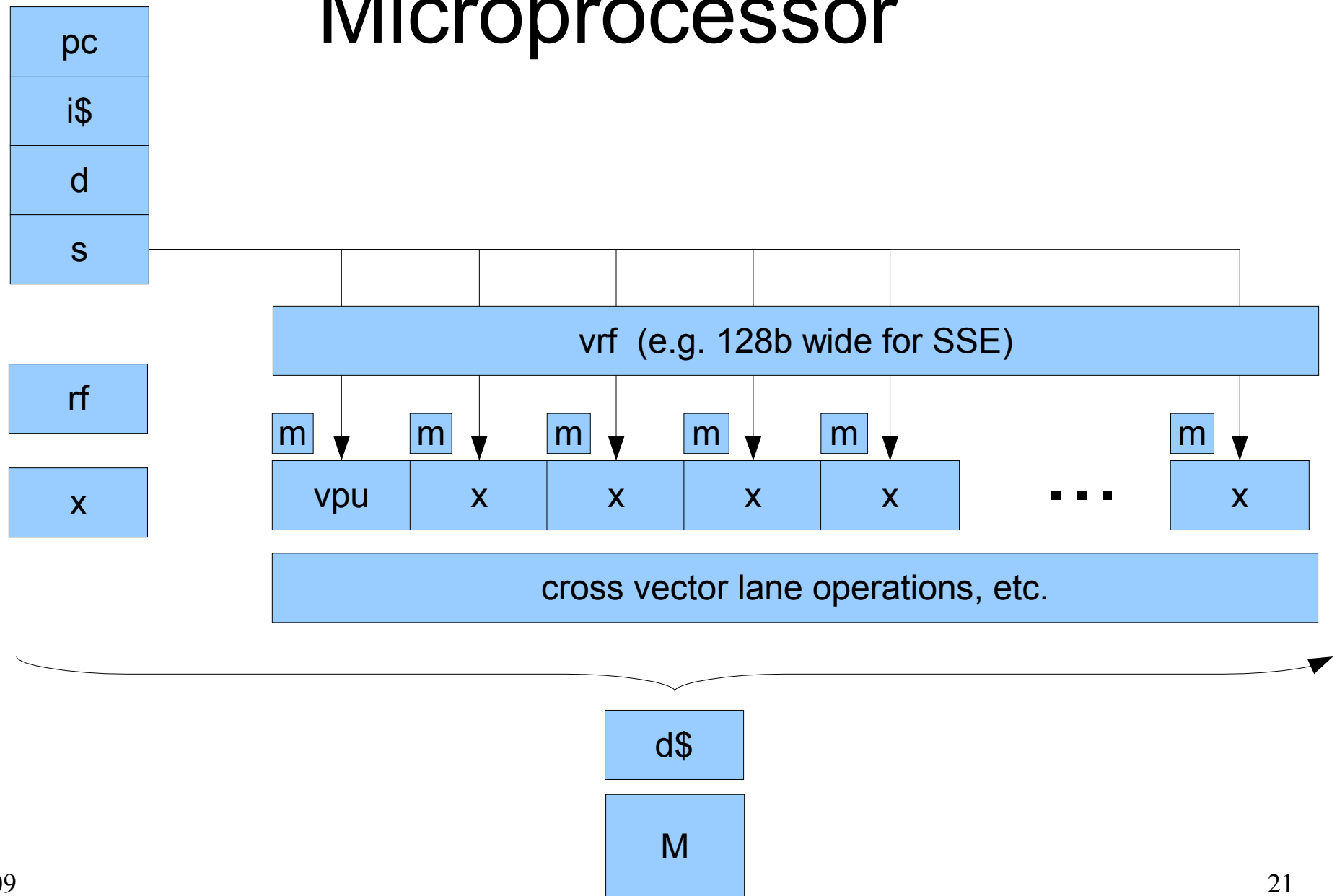
Scalar Processor



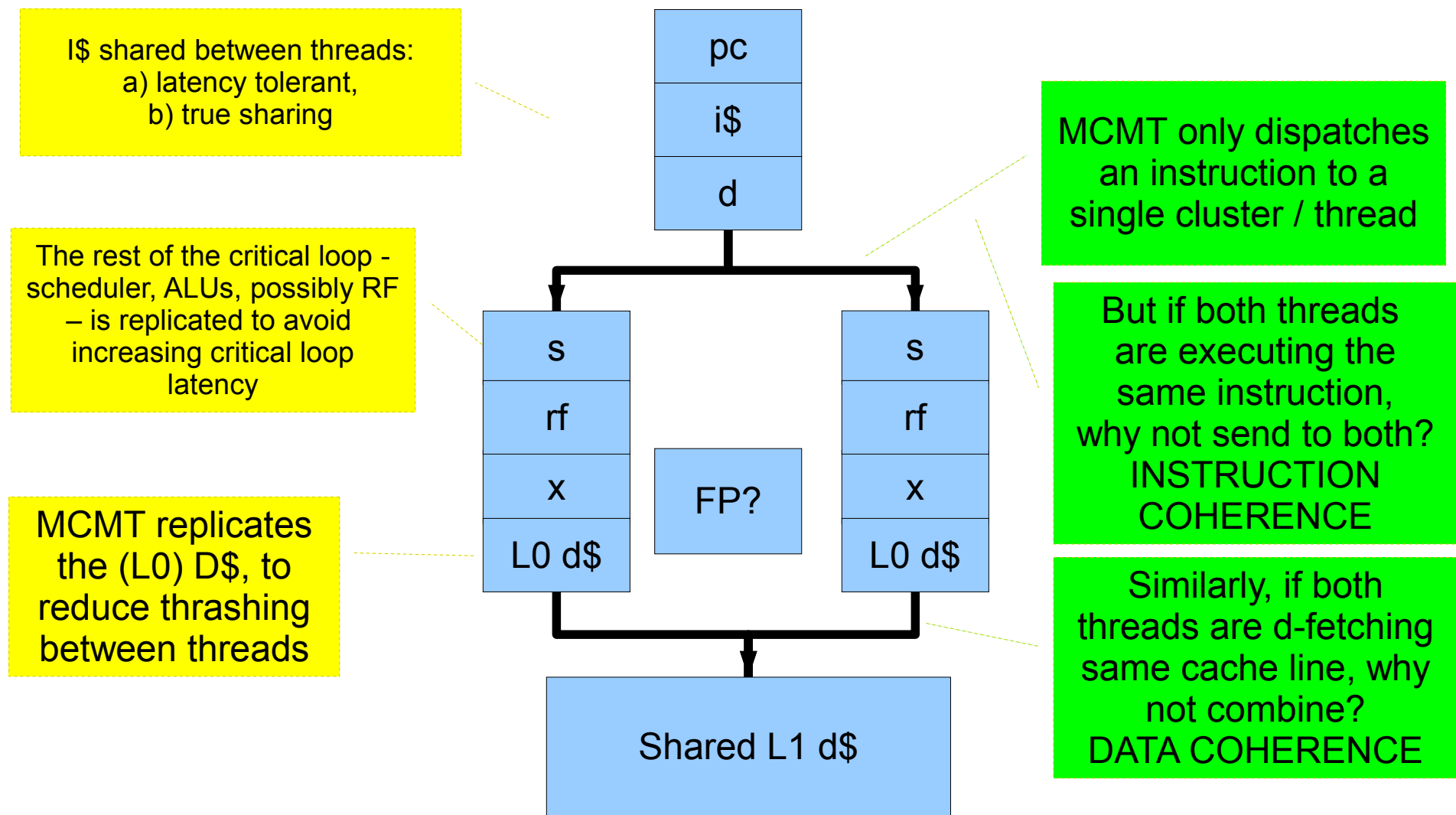
MIMD Parallel Processor



Vector/Parallel “SIMD” Microprocessor



Multicluster Multithreading



One Way of Looking at It

Vector Lane Threading
(SIMT, DIMT, NIMT)

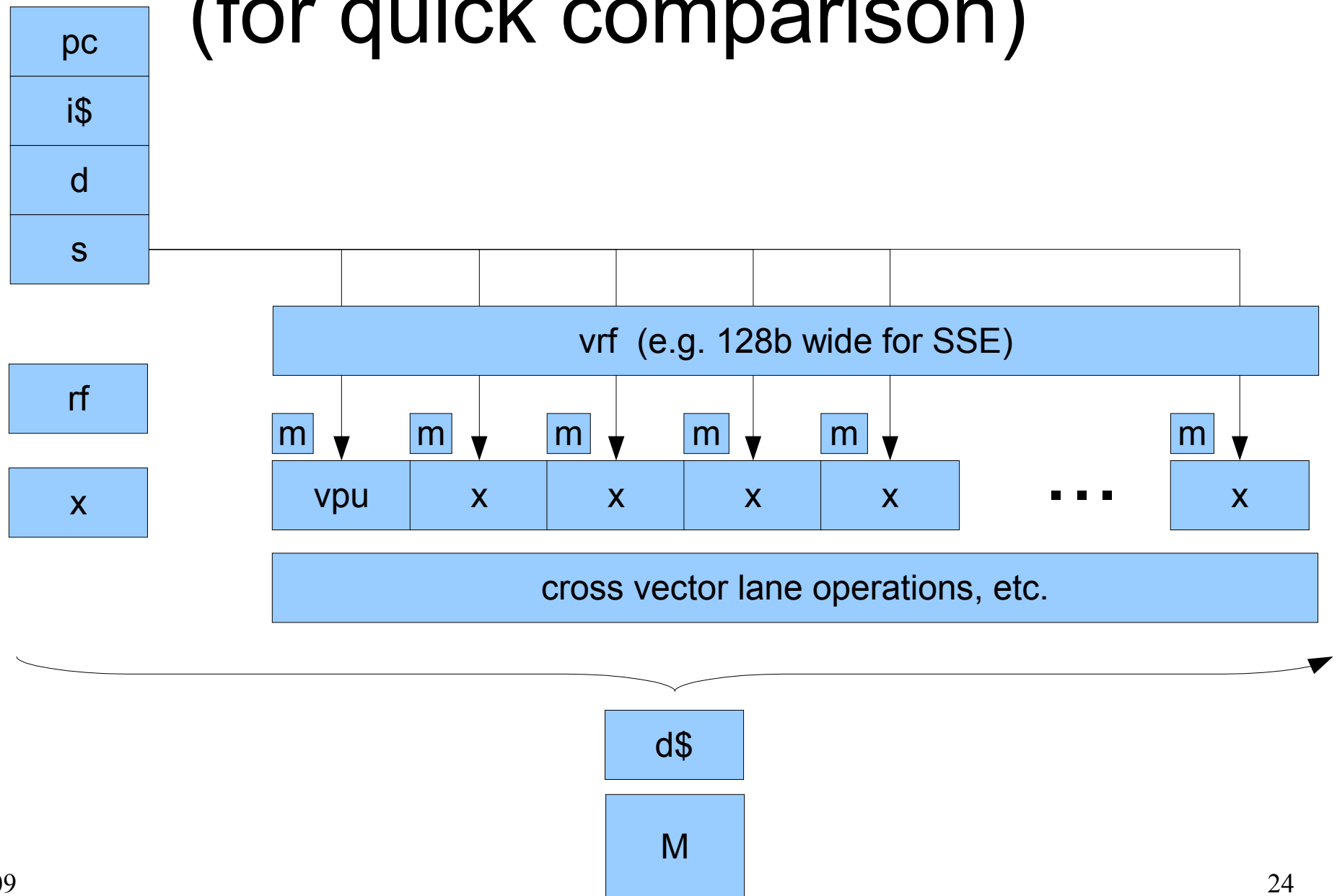
=

Multicluster Multithreading

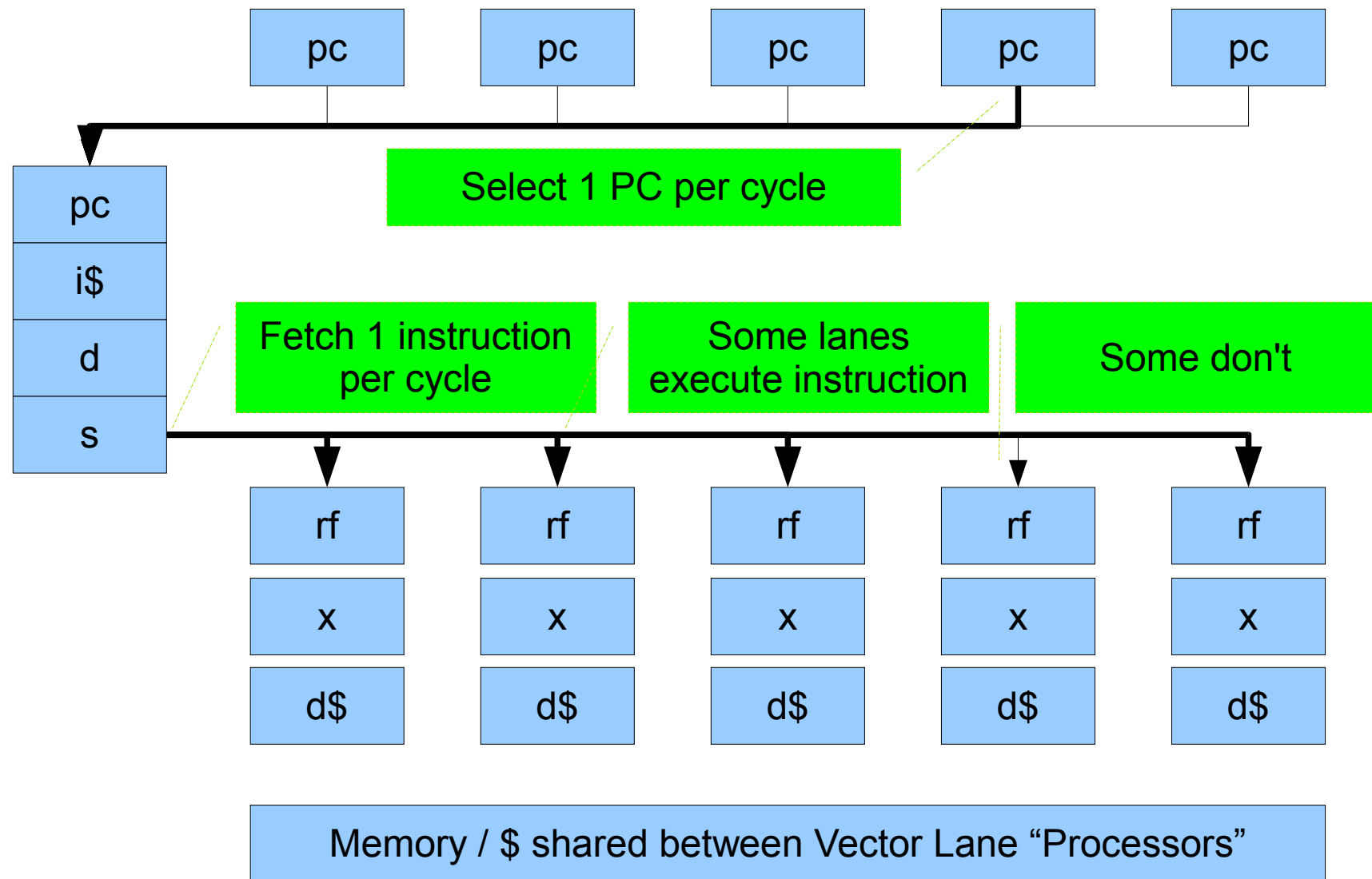
+

optimizations to take advantage
of Instruction Coherence
and Data Coherence

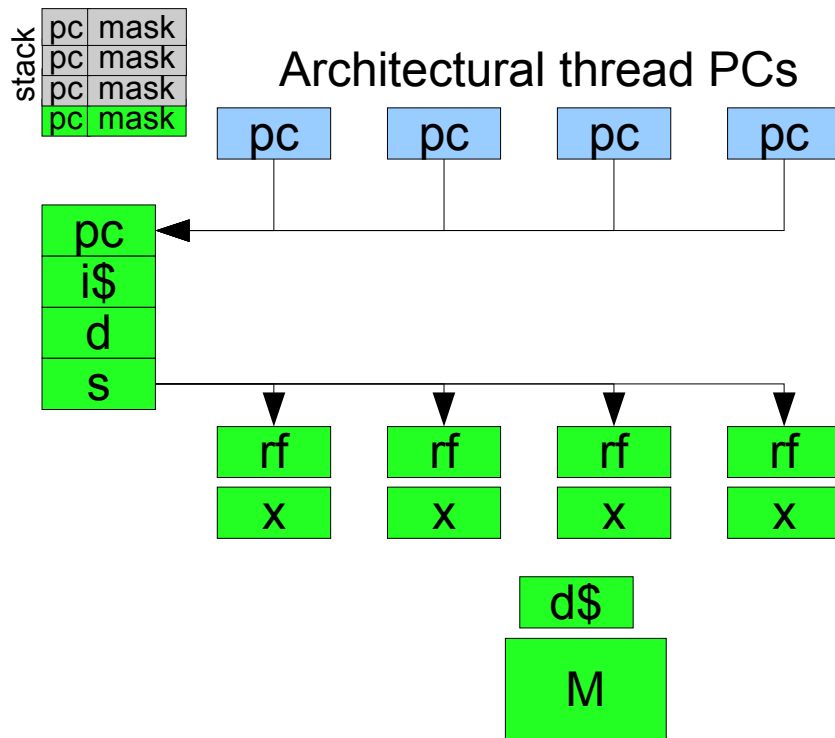
Vector/Parallel “SIMD” (for quick comparison)



Vector Lane Threading (aka SIMT)

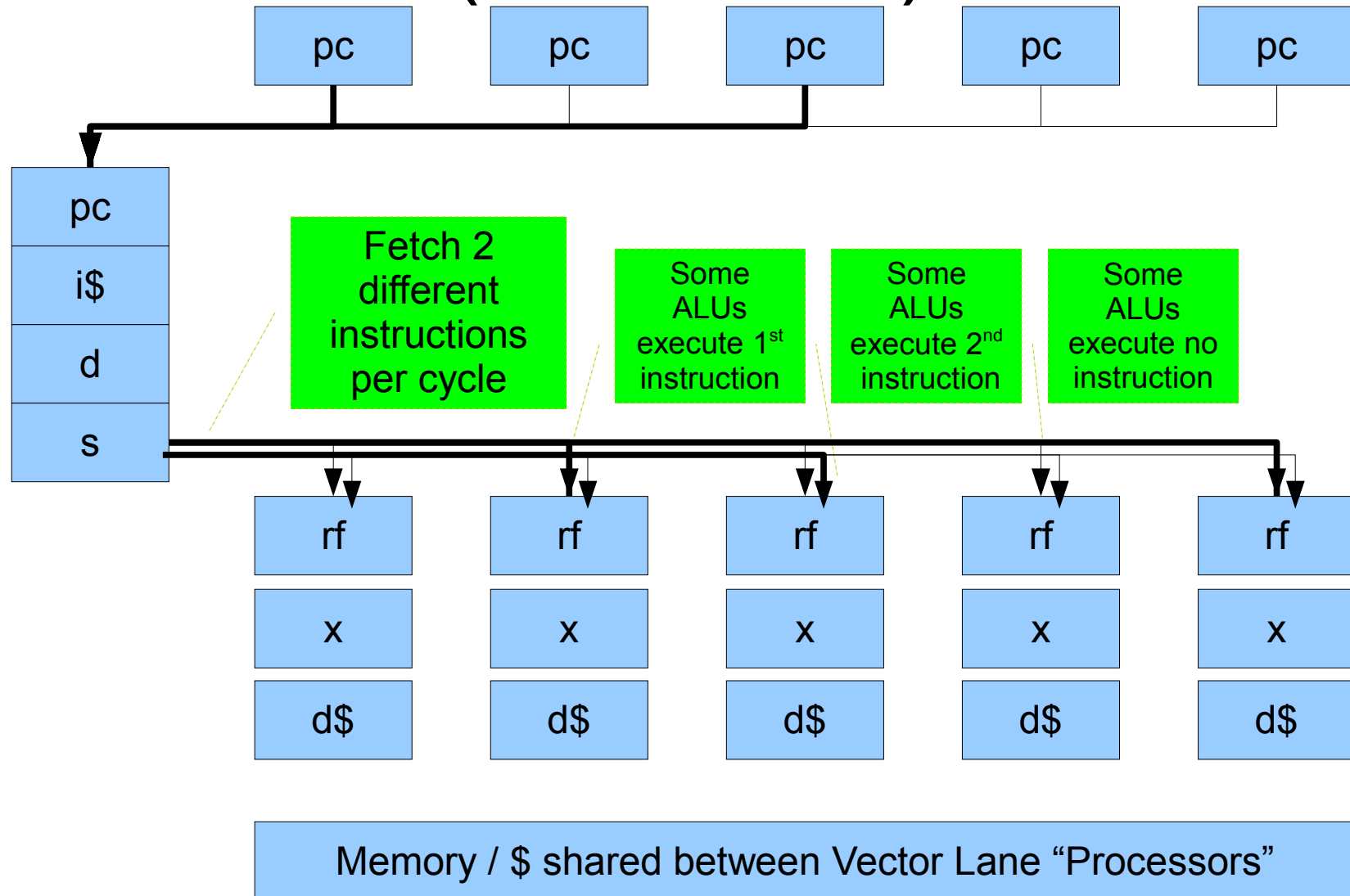


Classic SIMT Stack Algorithm



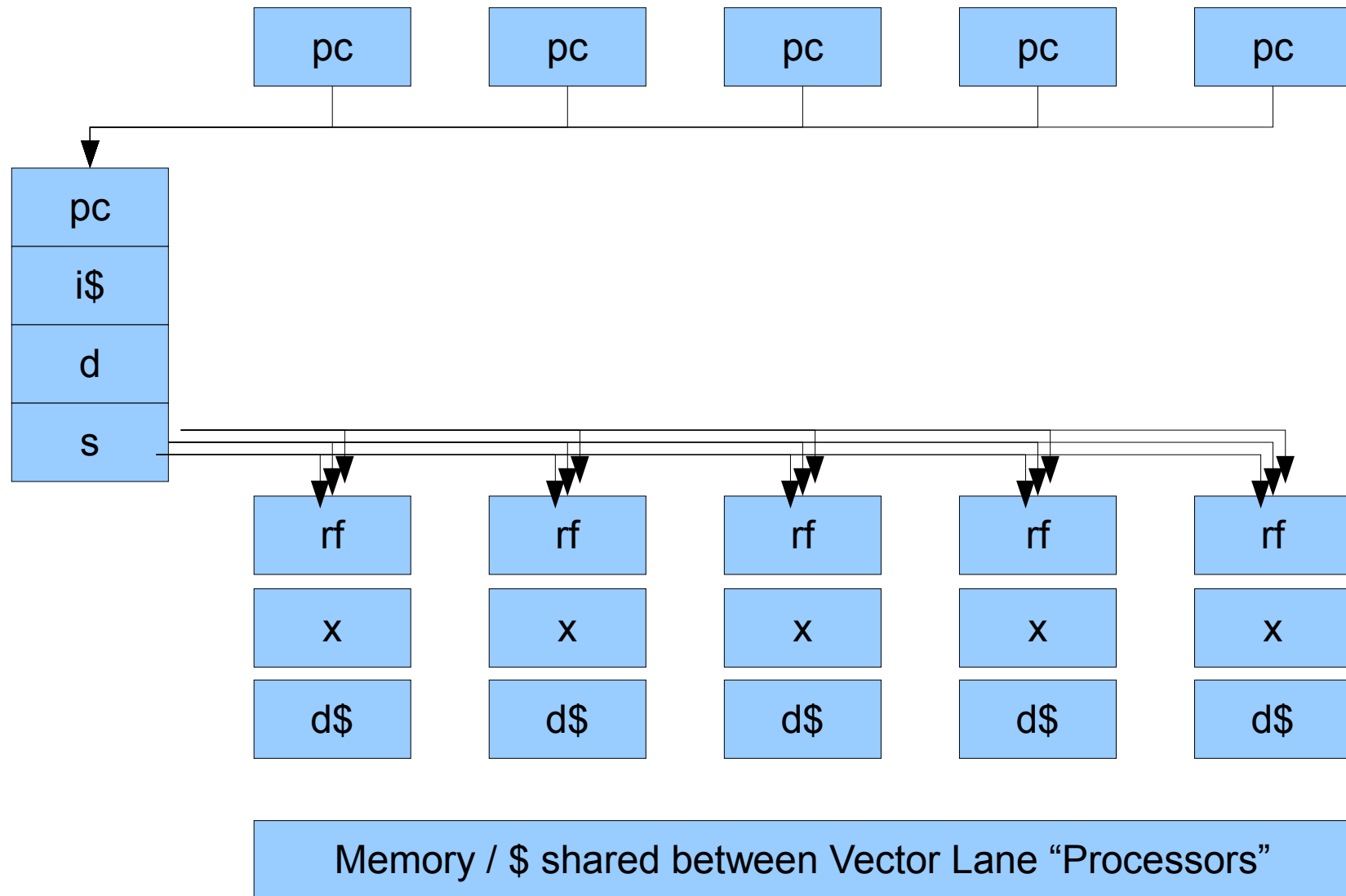
- Initially, all PCs in a warp are equal
- Execute PCs under mask
 - If diverging (IF)
 - Push target PC, mask
 - If converging (ENDIF, ENDLOOP, RETURN)
 - If matches TOS
 - Pop TOS.(PC,mask)
 - If switching (ELSE)
 - If matches TOS
 - Pop TOS.(PC,mask)
- Worst case:
 - 1 stack entry per vector lane
 - * number of conventional horizontal threads
 - But only one PC and one TOS entry need be examined at a time.
 - No CAMs.

Vector Lane Threading - 2 (aka DIMT)



Vector Lane Threading – N

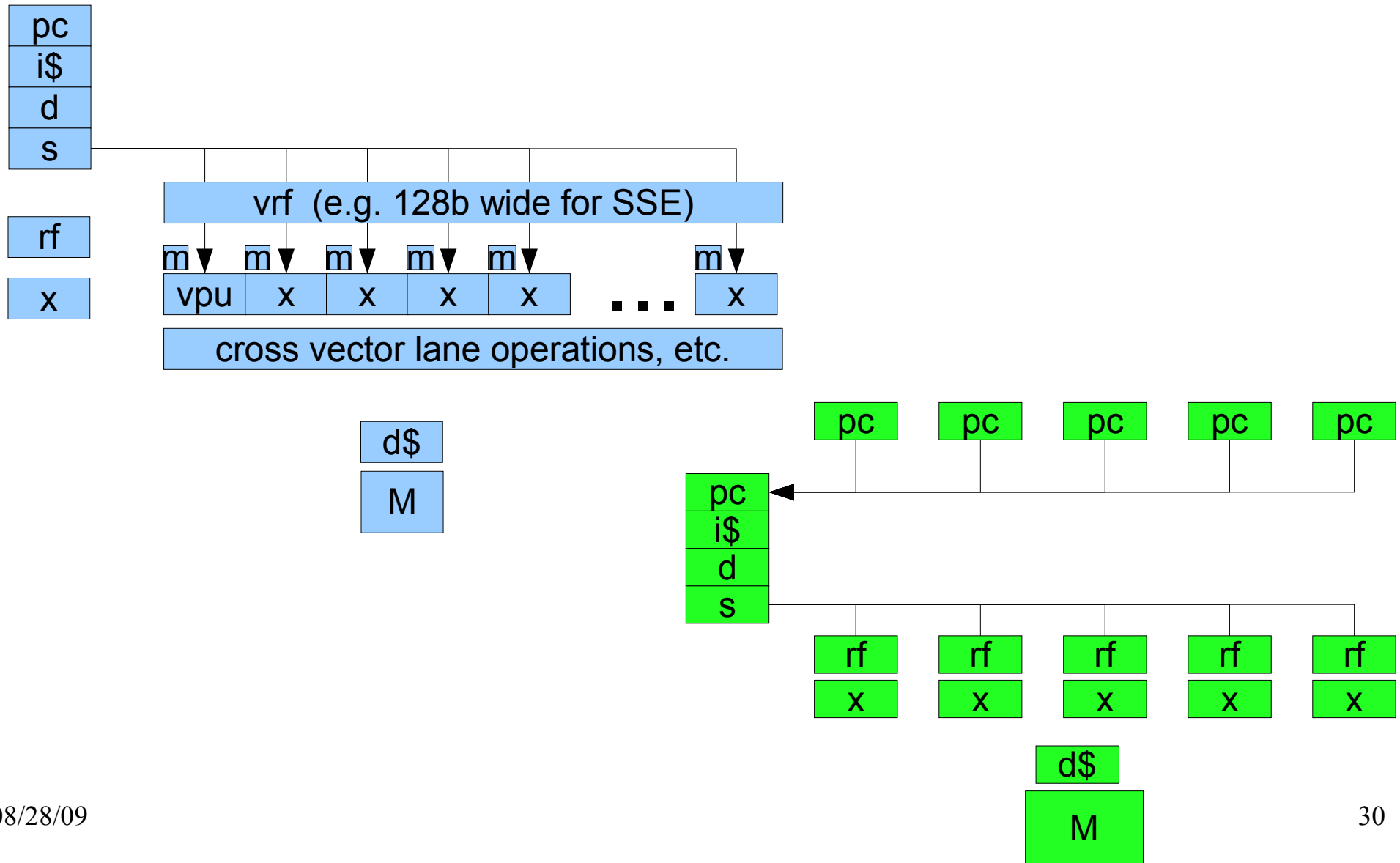
NIMT = N instructions, M threads, $N < M$



State of the Art

- Nvidia
 - Gt280: SIMT(Scalar), strictly scalar within lanes
 - Gt300: ?? MPMD, “MIMD-ish” ??
- ATI 7700
 - SIMT(VLIW): 8 lanes, lanes = 5 wide VLIW
- LRB
 - 512 bit (64B, 16x32b) SIMD

Comparing SIMD vector parallel to SIMT vector lane threaded



The Point of this Presentation

- Typical GPUs use
Vector Lane Threading, aka SIMT, aka SIMD
- Many people seem to believe this equivalent to
“conventional” vector processors
- NOT your Father's Vector Processor
 - SIMT is not just vectors with mask
and good scatter/gather
 - Reduces key sources of inefficiency in Parallel Vector Processors
- Vector Lane Threading is to Parallel Vector Processors
what Out-of-Order Execution is to VLIW
 - Microarchitecture, not Macroarchitecture
 - Same “ISA Concept” runs on scalar MIMD
 - Assuming you /encouraging you to have lots of threads

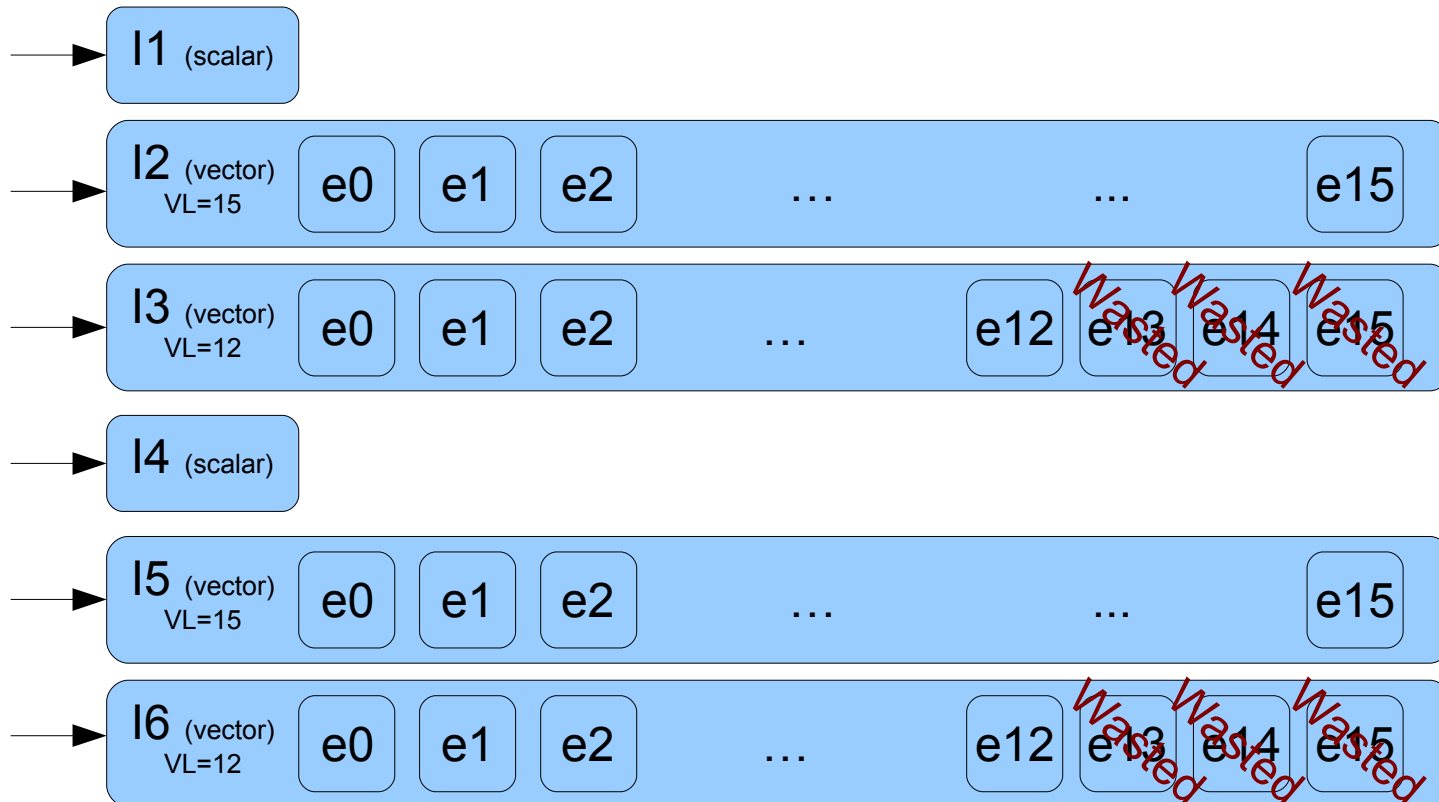
Inefficiencies in Parallel Vector Processors

- Vector Length
- Conditionals / Vector Masking
- Vector Lane Crossing

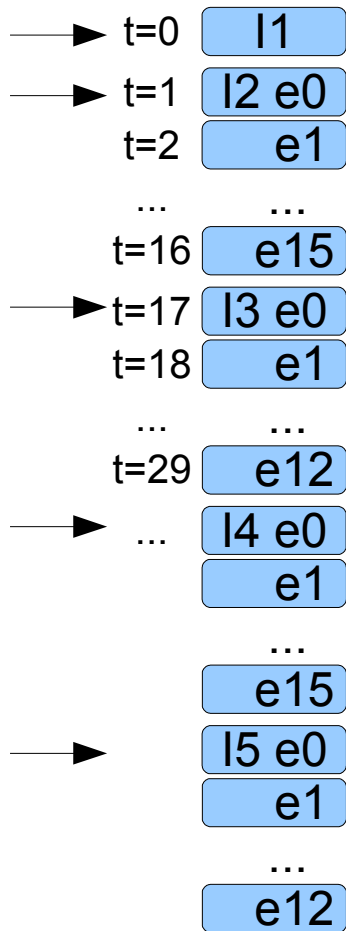
SIMT = no Vector Lane Crossing

- Vector Instruction sets always want “intra-lane” operations
 - Dot product: $V1_1 * V2_1 + \dots V1_N * V2_N$
 - Recurrences:
 - $A_1 = B_1$
 - $A_2 = B_1 + B_2$
 - $A_3 = B_1 + B_2 + B_3$
- Crossing vector lanes big source of complexity
 - Hardware complexity: wires, muxes
 - ISA complexity: instruction proliferation
- SIMT doesn't need vector lane crossing instructions, because different lanes are from different threads.

Parallel Vector Length Wastage



Vector Time Pipelining eliminates Vector Length Wastage

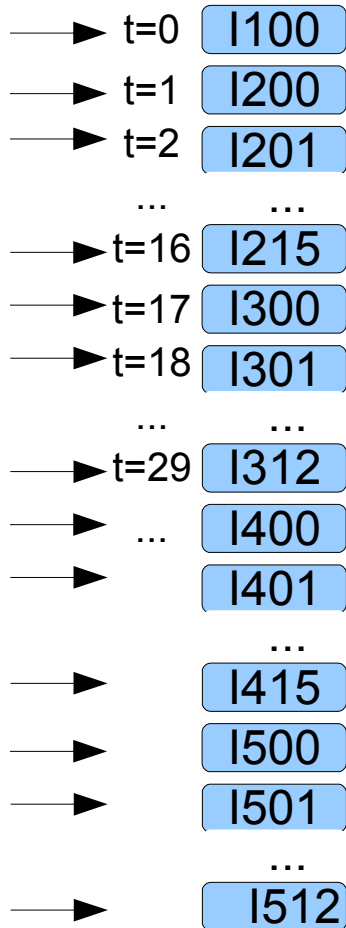


No wasted EU
- unused vector length skipped

Cost: latency

E.g. Cray-1 was NOT a parallel machine:
it was a pipelined vector machine
(with chaining)

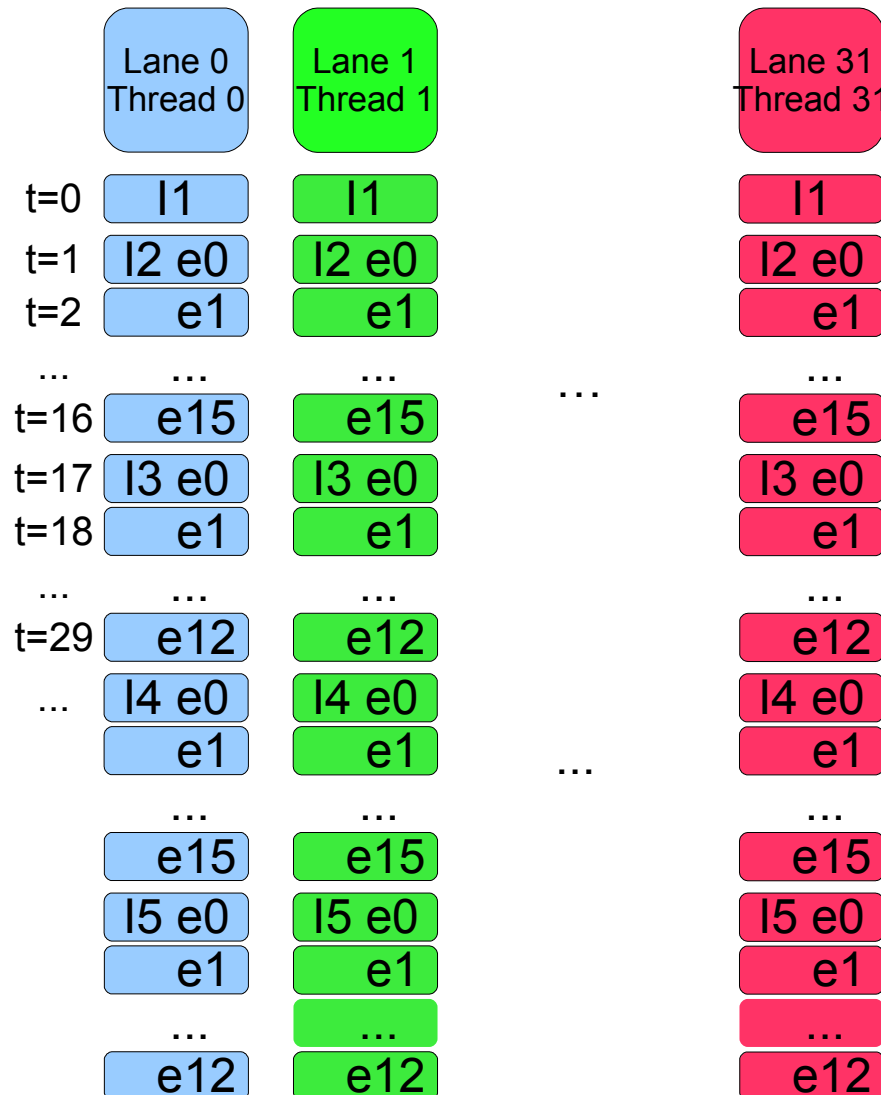
Scalar (and Scalar MIMD) also eliminate V_L wastage



No wasted EU
- unused vector length skipped

Cost: latency
+ instruction overhead

Vector Time Pipelining works with SIMT Vector Threading



Unclear how much existing GPUs do this.

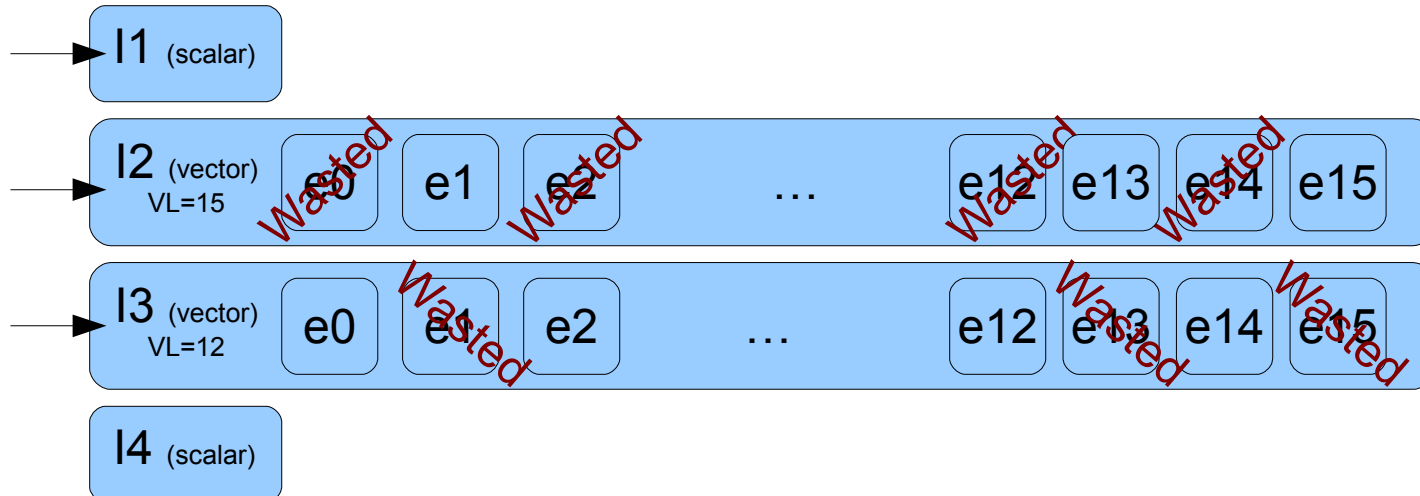
Apparently short vectors (2 or 4 ticks long.)

Many (but not all, not even most) of these slides amount to showing how **time pipelined vector instructions** on a **SIMT vector lane threaded microarchitecture** solve many utilization problems.

But don't worry, there's other neat stuff.

Vector Mask Wastage

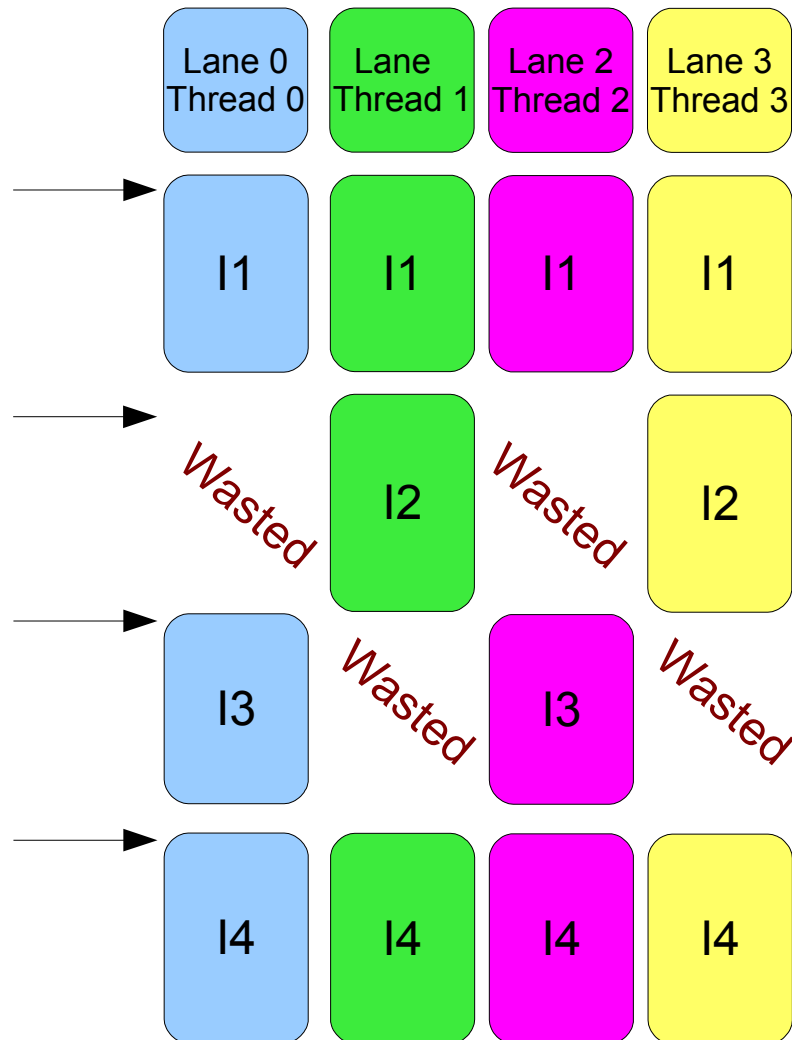
```
I1
for all i
  if odd then I2 else I3
I4
```



Scalar Simple SIMD

Conditional Wastage

```
I1
for all i
  if odd
    then I2
  else I3
  endif
end loop
I4
```



Many people say...

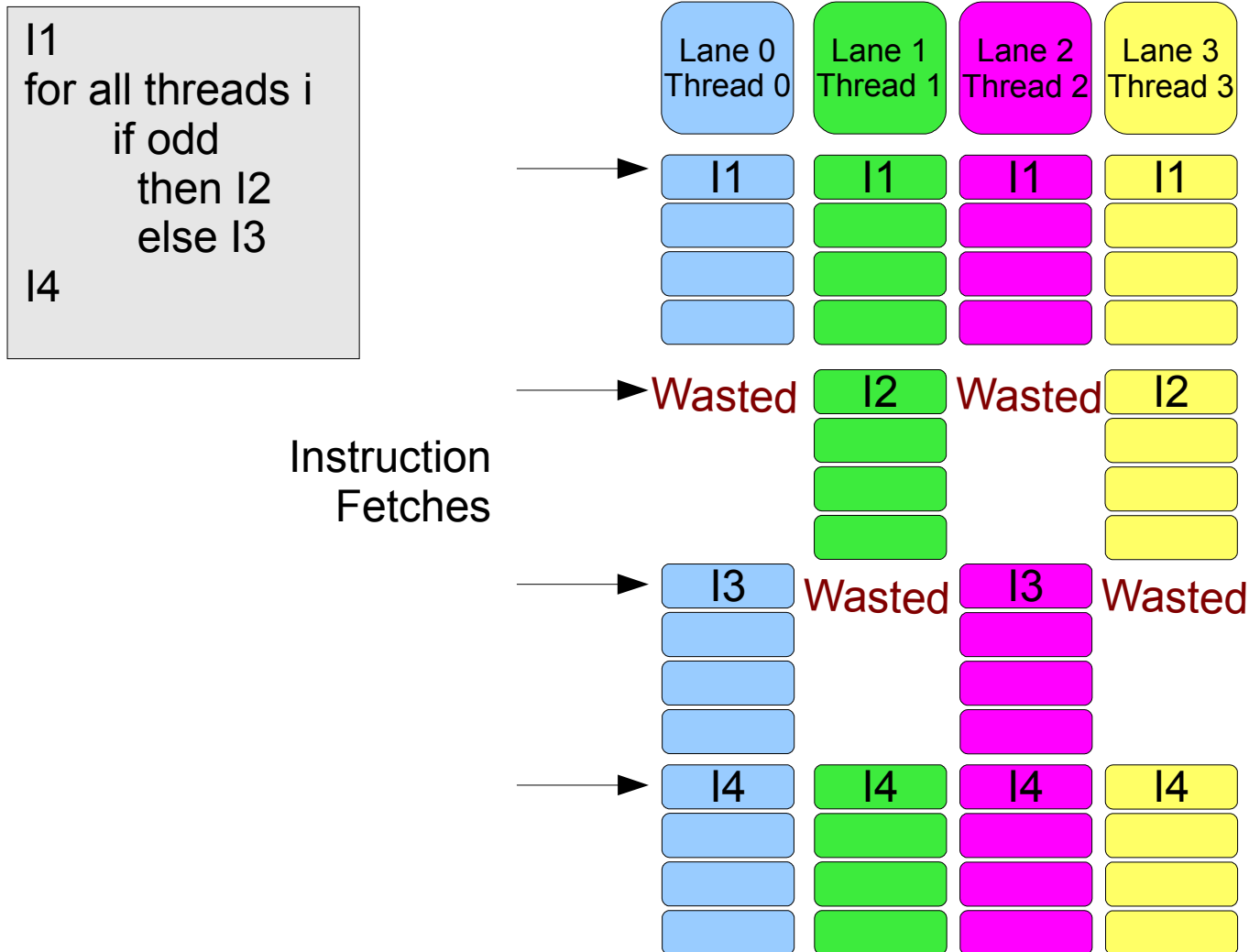
Doesn't this mean
that SIMT/SIMD/Vector Lane Threading

Is no more efficient than
Width Parallel Vectors with Vector Masks
?

Hell No!

There are lots of ways to reduce
SIMD conditional wastage

SIMD Conditional Wastage with Time Pipelined Vectors



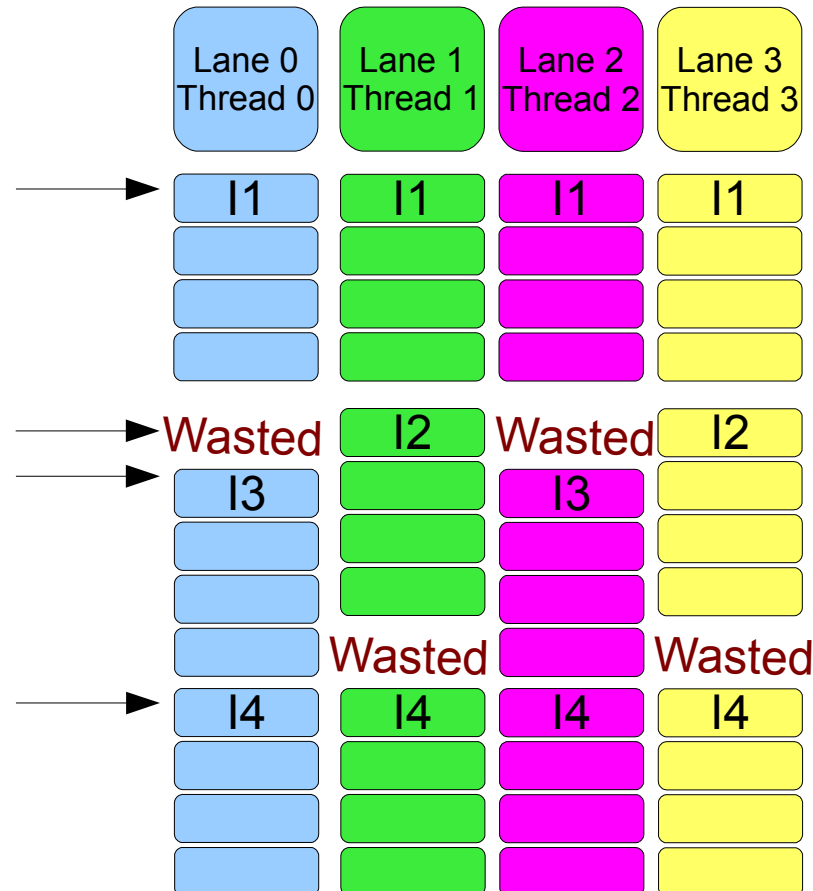
Reduced SIMD Conditional Wastage with Time Pipelined Vectors

```

I1
for all threads i
  if odd
    then I2
    else I3
I4
    
```

*Why not start instruction
fetch + start instructions on
idle lanes while busy
executing multicycle time
pipelined vector
instructions?*

(See RealWorldTech.com)



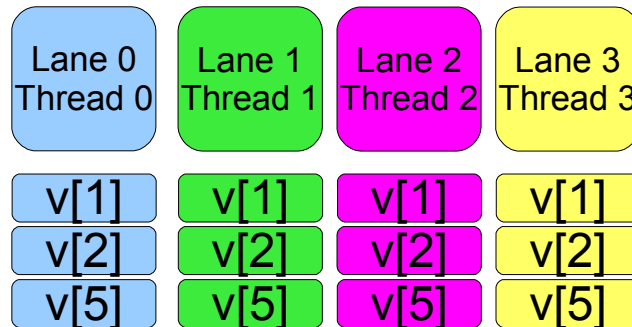
Time Pipelined Vectors also help Vector Masking within Lane

```
l1
for all threads i
  for all vector elements j
    if cond(j) then
      v[j] = ...
```

l4

Where $\text{cond}[j] =$
arbitrary, e.g.
 $\{0, 1, 1, 0, 0, 1, 0\}[j]$

Assuming $\text{cond}[j]$
is available to
instruction sequencer

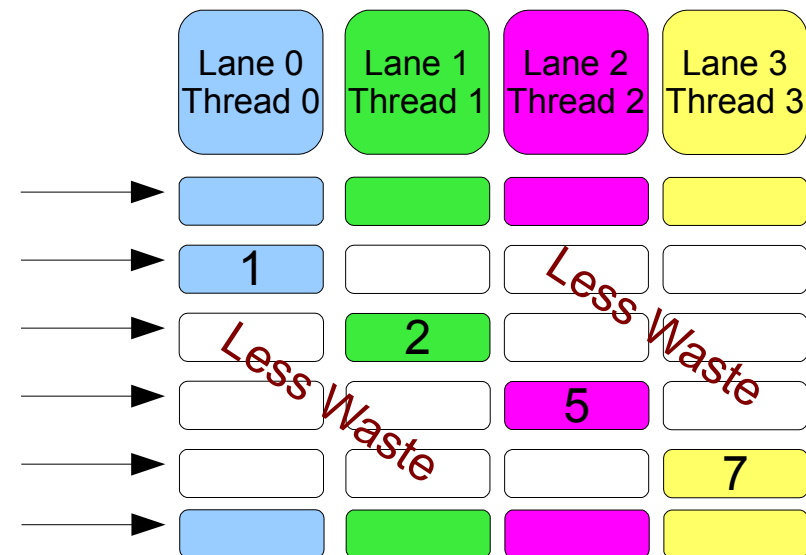
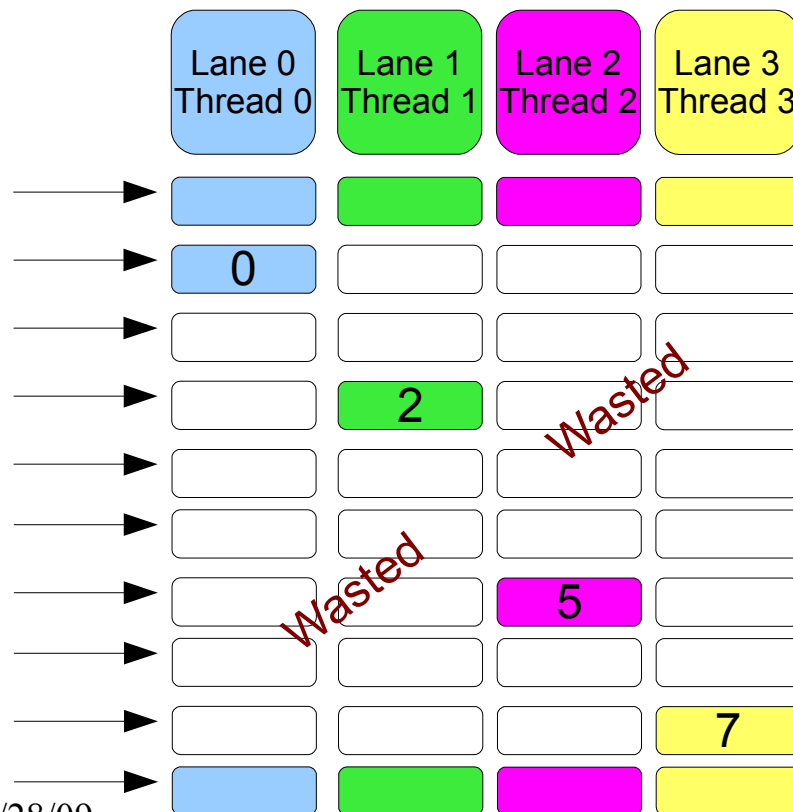


That was easy...

But let's do something harder
where the “SIMD” threads
take different paths

Predication executes all paths; SIMT only executes instructions used at least once

```
switch(f(i)) {  
  case 0: I0; break;  
  case 1: I1; break;  
  ...  
  case 7: I7; break;  
}
```



Why not a loop buffer at each lane?

```

I0
for all threads I
  repeat 100
    for all vector elements j
      I1: if cond[i,j] then
        I2: v[j] = ...

```

I3

Where $\text{cond}[i,j] =$
arbitrary, e.g.

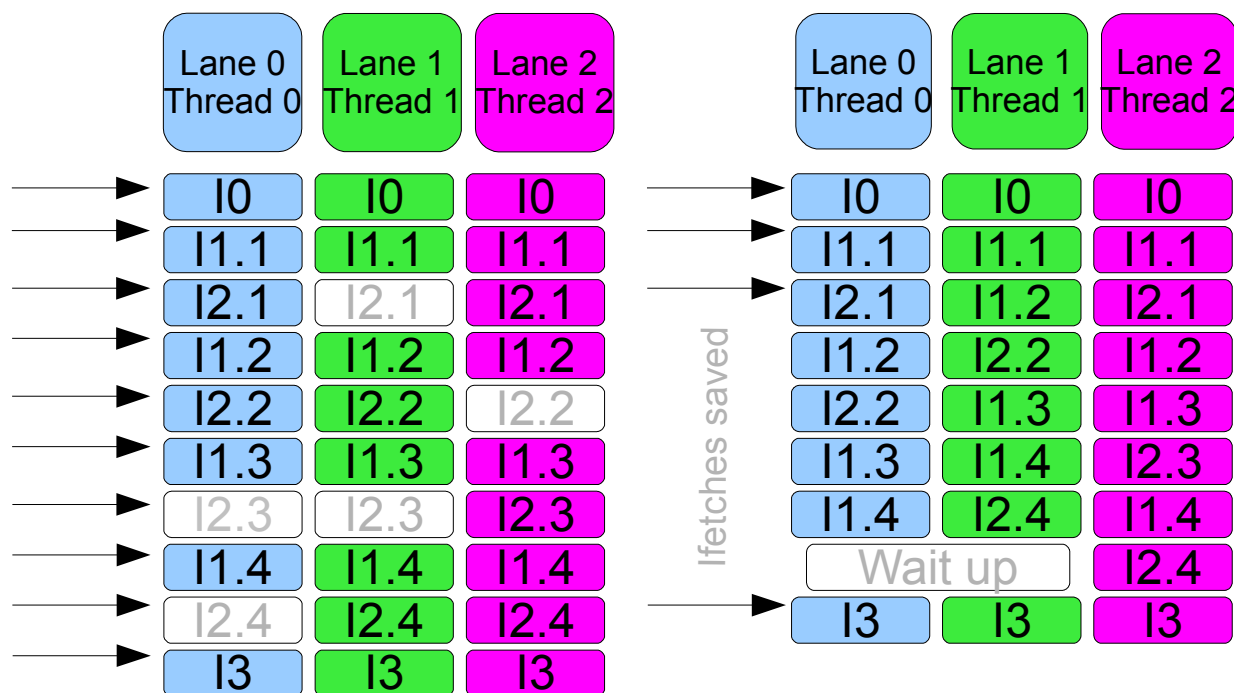
101

110

000

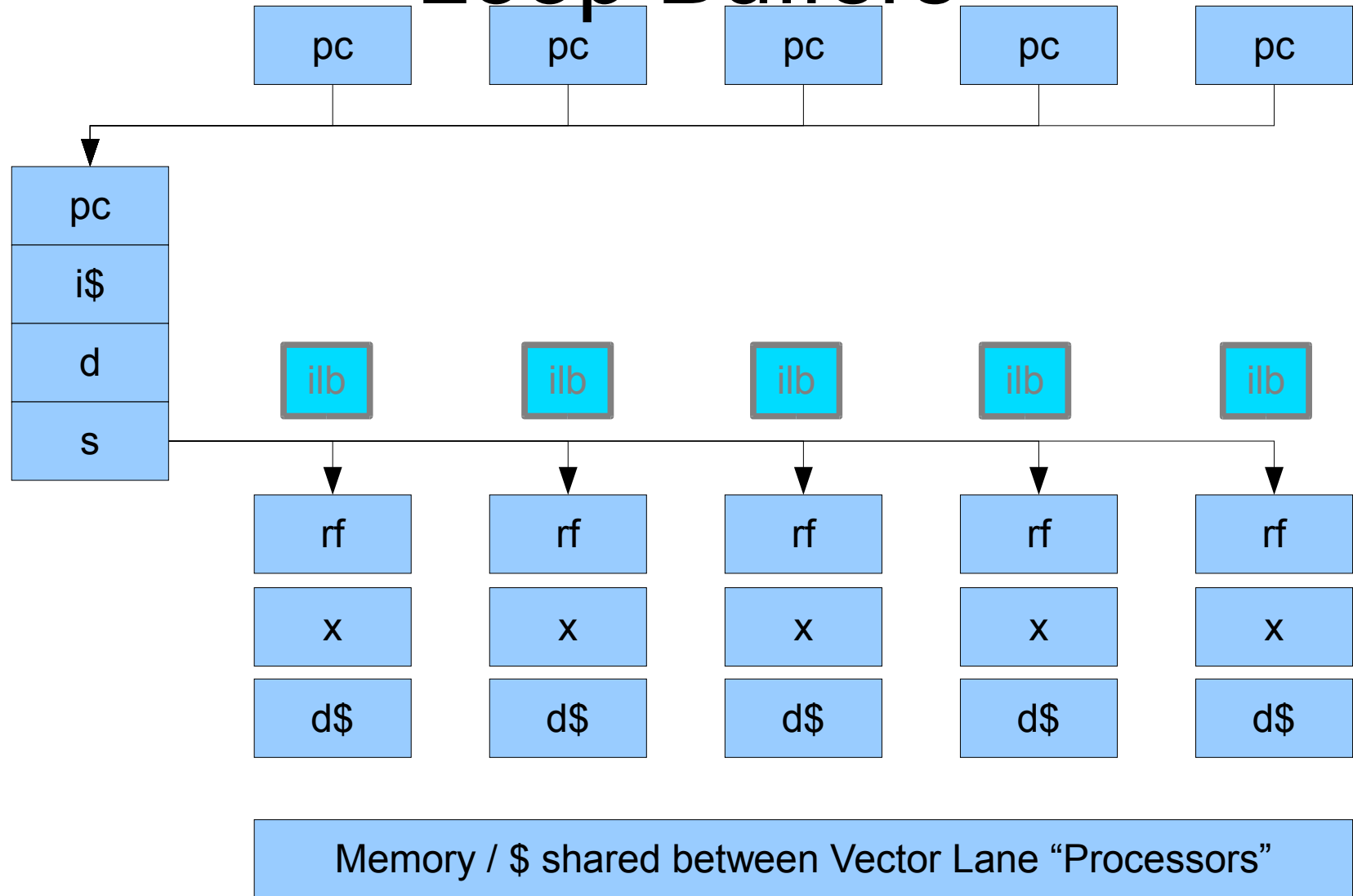
011

Assuming $\text{cond}[i,j]$
is NOT available to
instruction sequencer



Vector Lane Threading

Loop Buffers



SIMD → MIMD spectrum



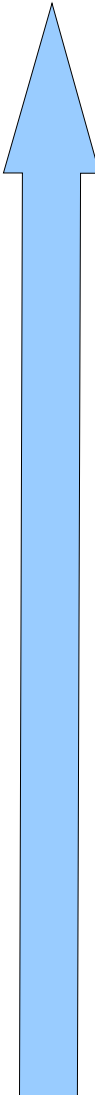
More stuff replicated
per lane

- Instruction buffer
- Loop Buffer

...

Less SIMD idleness

More decoupled the
processors



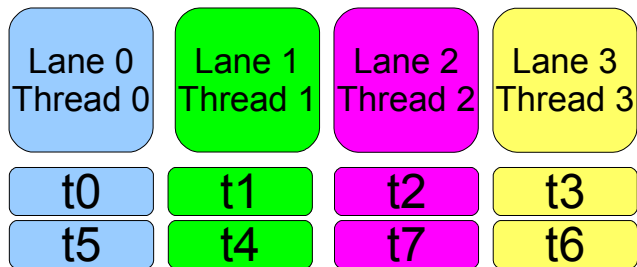
More coupled the
processors

More area/power
saved

More stuff shared

Rejiggering Threads between Cycles of a multi-cycle Warp

Original, Imbalanced = 50% utilized

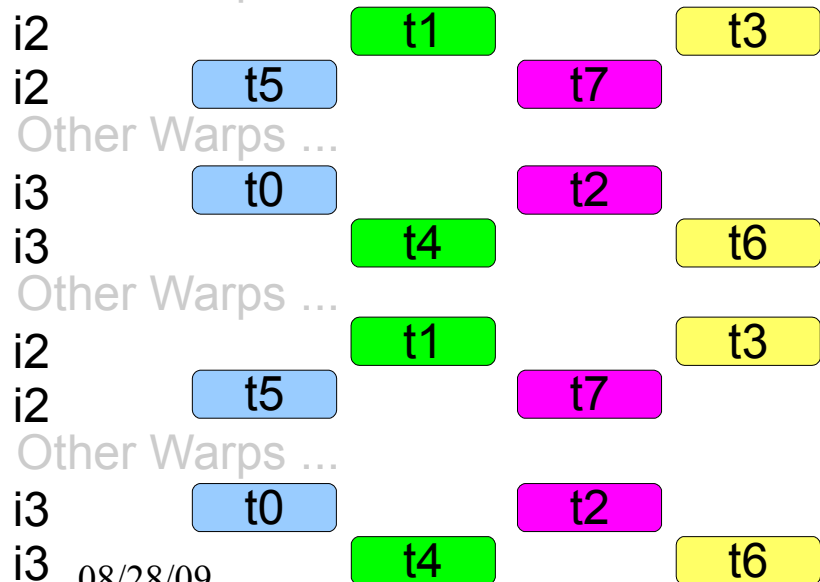


Example assumes
Thread skewing
between waves
in a wavefront

```

l1
repeat i=0 to ...
  for threads t=0 to 7
    if even(i) XOR even (t)
      then l2
    else l3
l4
    
```

Other Warps ...



08/28/09

Rejiggered within Wavefront = 100% utilized



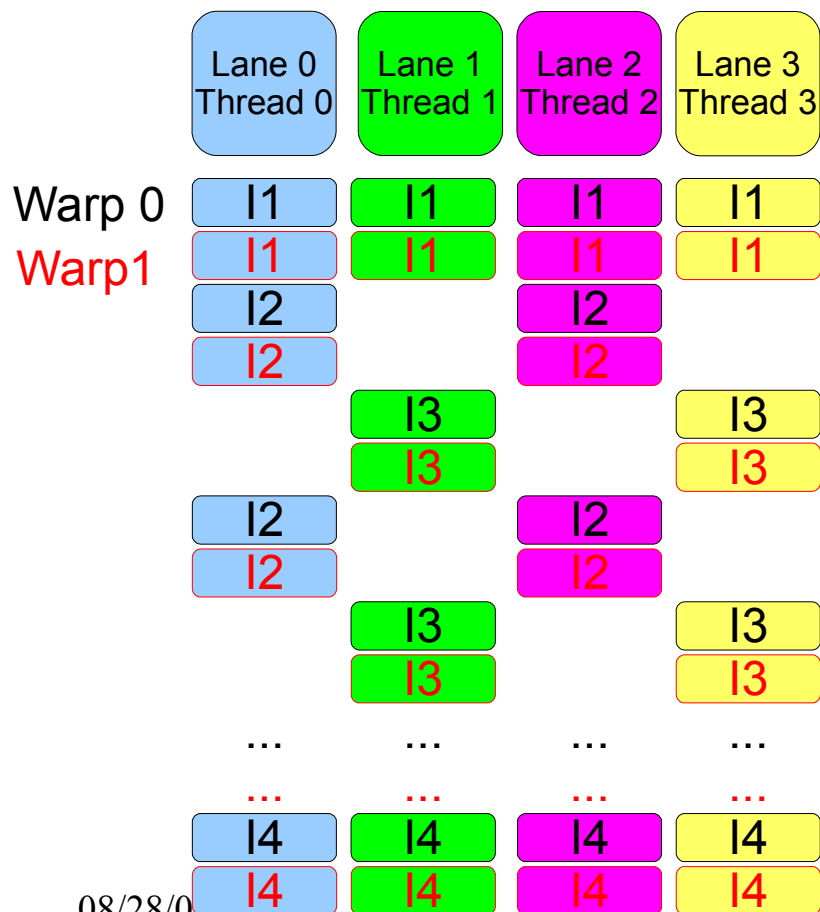
Other Warps ...

At least 4 levels of thread/warp rebalancing

- Rejiggering within a warp (wavefront, multicycle SIMT thread group) between cycles (phases, waves) of same warp
 - Fairly easy – just a few more bits / logic on mask
 - Migrating (exchanging) threads between warps but staying in same lane
 - Fairly easy – more bits
 - Migrating threads between lanes of same warp
 - Hard – involves copying registers.
 - Probably not worth doing, except for RF lane tricks
 - Rebuilding warps (wavefront, multicycle SIMT thread group) completely
 - Hard – involves copying registers
- 08/28/09 – Probably only worthwhile after really long latency events.

How about Rebalancing Threads between Warps?

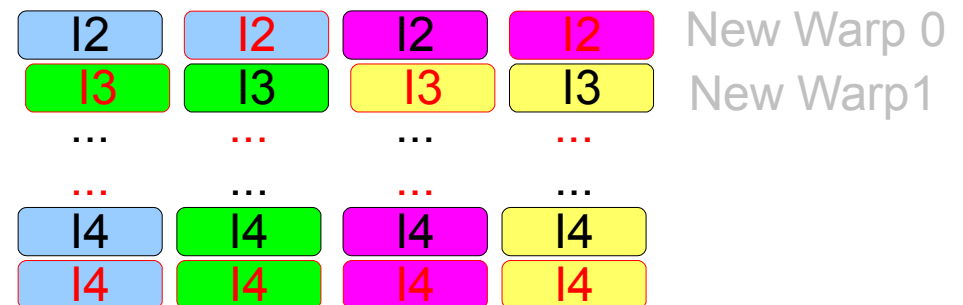
Original, Imbalanced



```

I1
repeat i=0 to ...
  for warps w=0 to 1
    for threads t=0 to 3
      if even(t+i)
        then I2
      else I3
I4
    
```

Rebalanced



How to Rebalance Threads between Warps

- Compiler should group similar threads as much as possible
 - Hints to thread scheduler
 - But such static balancing can only go so far
- Dynamic balancing
 - Moving threads between warps within the same lane-aligned register file looks doable (change of pointer) (HW?)
 - Moving threads between lane RFs involves copying => Ucode? Software? SMs?

Remember Interleaved vs. SoEMT Threading within Lanes?

Original, Imbalanced

	Lane 0 Thread 0	Lane 1 Thread 1	Lane 2 Thread 2	Lane 3 Thread 3
Warp1	1	1	1	1
Warp2	2	2	2	2
Warp3	3	3	3	3
Warp1	1	1	1	1
Warp2	2	2	2	2
Warp3	3	3	\$miss	3
Warp1	1	1	1	1
Warp2	2	2	2	2
Warp4	4	4	4	4
... wait till Warp 3 miss returns replaces Warp 1 in running list				
Warp3	3	3	3	3
Warp2	2	2	2	2
Warp4	4	4	4	4

Minimum latency for data dependencies hidden by IMT (Interleaved Multithreading)

Long latencies, esp. unpredictable long latencies (TLB miss, cache miss) hidden by SoEMT

Option 0: block all threads in warp if any misses

Option1: split warp to allow unblocked threads to run

Little Known Fact

- I developed the P6-style RS in part for vector scatter/gather
 - Allow earliest elements to complete ASAP
 - Instead of waiting for all pieces to arrive
- Problem: once vector is split up into elements hard to regain efficiency of controlling all vector at a time.
- Vector threading makes this easy.
Key: PC

- Literally, the PC (per vector lane thread) is the “key” as in database join key – the thing that tells you how to get threads back into alignment.

Bottom Line

- There's a hell of a lot that you can do to avoid SIMD wastage vs. Parallel Vector Wastage
 - Time pipelined vectors in lanes
 - Overlapping different multicycle instructions
 - Loop buffers per lane
 - Rebalancing, e.g. after gather
- Unclear which are done in present GPUs
 - Most seem to be
 - Some are in recent dissertations

Summary of SIMT VPU Utilization Improvements

- SIMT inherently and fundamentally is more efficient, filtering out null masks. (Minor)
- SIMT + vector time pipelining
 - More efficient masking and VL (Major)
 - Extra instruction fetch opportunities during multicycle vector operations. (Major)
 - Also: extra instruction fetch opportunities during multicycle wavefronts (Major)
- SIMT rebalancing
 - Rejiggering within warp: same lane, different cycles (Major)
 - Migrating between warps at same lane (Minor)
 - Rebuilding warps (Major, but complex)

Bottom Line²

- SIMT / SIMD / Vector Lane Threading
- Cheap microarchitecture for threads that have similar (but not identical) control flow
 - Shares front end ...
- Same “ISA” (conceptual) can run on MIMD, if that is better for workload
 - i.e. SIMT / SIMD / Vector Lane Threading is microarchitecture, not macroarchitecture
 - Can run any reasonable instruction set within vector lane
- Nice: ENDIF, ENDLOOP convergence indications

Bottom Line³

- SIMT / SIMD / Vector Lane Threading
 - Encourage programmer to have lots of lightweight threads
- GLEW OPINION:
 - It is easier to write programs with 1000s of threads than with 10s of threads
 - PRAM style programming
- These are NOT just vector machines.

Top 10 reasons why vector lane threading (aka SIMT, aka NIMT) is better than 16 wide SIMD vector parallel

1. **A SIMT pipeline can trivially run SIMD vector parallel code, but not vice versa.**

2. If **vector ALU utilization** is an issue, then SIMT probably has (or can be built to have) significantly better vector ALU efficiency. SIMT is less vulnerable to problems due to vector length and vector mask. This is explained in the slides.

3. **SIMT is less dependent on software development.**

You can change the “vector width” of a SIMT microarchitecture, and all existing code continues to run, usually still efficiently. Whereas if you change, e.g. from 512 bit vectors to 1024 bit vectors, SIMD parallel vector software needs to be rewritten.

4. SIMD vector parallel ISAs always slide down the slippery slope of cross **vector lane instructions**. First you want inner product, then recurrences, then... SIMT doesn't have this temptation, since the different lanes are from different threads.

5. **SIMT seems to be a better match for software.** I must admit that I was surprised when I learned this, since I'm a vector guy from way back. But it seems to be the case. **DirectX's API is basically a SIMT API.**

You need a good compiler to convert from SIMT DirectX to SIMD parallel vectors.

6. SIMT may be considered just a step along the road to **MIMD**.

7. There are significant **power savings opportunities** due to time pipelining vectors.

I.e. SIMD parallel vectors waste power, because they do not take advantage of the correlation in data patterns between successive vector elements.

While it is possible to build a non-SIMT vector pipeline that takes advantage of this, it is not SIMD vector parallel. E.g. it is vector time pipelined, with chaining. E.g. instead of a 16 wide vector machine, you might build a vector machine that takes 4 cycles to execute a vector instruction on 4 ALUs, which keeps 4 such instructions in flight, chaining into each other to keep each other busy.

I've worked on machines that did such chaining. (Gould). In fact, I invented the P6 RS as a chaining control unit. They are complex.

SIMT has much simpler pipeline control than vector chaining.

OK, so that's only 7 reasons. More will be coming.

Backup

To Learn More

- David Kanter, Real World Tech article:
 - <http://realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=12>
 - It is by no means completely accurate. For example, I don't think that the author understands the potential of SIMT - he thinks that SIMT will just evolve back to SIMD. He thinks that the "ensemble" intra-warp instructions are a step like this, whereas I see these as being much more like multi-thread barrier instructions.
 - Nevertheless, it is the best I have found so far. It is good enough that Nvidia folk have started referring programmers to this article, rather than Nvidia internal documents.
- UIUC GPU course, Wen-Mei Hwu:
 - <http://courses.ece.illinois.edu/ece498/al/>
- Wilson Wai Lun Fung, UBC MS thesis
 - https://circle.ubc.ca/bitstream/2429/2268/1/ubc_2008_fall_fung_wilson_wai_lun.pdf
 - Dynamic Warp Formation: Exploiting Thread Scheduling for Efficient MIMD Control Flow on SIMD Graphics Hardware

Covering My Ass

- This presentation is NOT about Larrabee
- But folks often ask for LRB references:
 - **Larrabee: A Many-Core x86 Architecture for Visual Computing.**
Larry Seiler¹, Doug Carmean¹, Eric Sprangle¹, Tom Forsyth¹, Michael Abrash², Pradeep Dubey¹, Stephen Junkins¹, Adam Lake¹, Jeremy Sugerman³, Robert Cavin¹, Roger Espasa¹, Ed Grochowski¹, Toni Juan¹, and Pat Hanrahan³
ACM Transactions on Graphics, Vol. 27, No. 3, Article 18, Publication date: August 2008.
http://download.intel.com/technology/architecture-silicon/Siggraph_Larrabee_paper.pdf
 - Heck, LRB has a wikipedia page!

Academic Work: Fung

- Wilson Wai Lun Fung, UBC MS thesis
 - https://circle.ubc.ca/bitstream/2429/2268/1/ubc_2008_fall_fung_wilson_wai_lun.pdf
 - Dynamic Warp Formation: Exploiting Thread Scheduling for Efficient MIMD Control Flow on SIMD Graphics Hardware
- “we propose dynamic warp formation and scheduling, a mechanism for more efficient SIMD branch execution on GPUs. It dynamically regroups threads into new warps on the fly following the occurrence of diverging branch outcomes. We show that a realistic hardware implementation of this mechanism improves performance by an average of 47% for an estimated area increase of 8%.”
- MIMD=1; NREC SIMD = 0.18; PDOM SIMD = 0.7; DWF SIMD = 0.9
 - PDOM best on Black-Scholes (=MIMD, DWF=0.85x). DWF better elsewhere.
 - BS: transcendental subroutines with branches
- PDOM: MIMD=0.26, 8-wide=0.21, 16-wide=0.19, 32-wide=0.16
- Real program: NREC=0.47, PDOM=0.83, Oracle=0.84
- Register lane aware dynamic warp formation
- Warp scheduling policies: Time Stamp, Program Counter, prefer Majority, prefer Minority, PDOM priority
 - Majority usually best; sometimes PC or PDOM priority

Academic Work: Kozyrakis et al, Vector Length Threading

- Kozyrakis et al, Vector Length Threading
 - Christos Kozyrakis (Stanford prof) was very happy to hear that I was calling SIMT/DIMT/NIMT vector length threading, generically, since he has published a paper by that name:
 - Citation: Suzanne Rivoire, Rebecca Schultz, Tomofumi Okuda, Christos Kozyrakis, "Vector Lane Threading," icpp, pp.55-64, 2006 International Conference on Parallel Processing (ICPP'06), 2006
 - Unfortunately, their use of the term "VLT" differs from my use:
 - Abstract excerpt: "we propose vector lane threading (VLT), an architectural enhancement that allows idle vector lanes to run short-vector or scalar threads."
 - I like the term "vector length threading" because it accurately describes my understanding of SIMT/DIMT/NIMT.
 - I am funny like that: I like my composite terms to have accurate meanings. I will not change my usage of this term at this time. I think the confusion is minor.
 - Their VLT paper emphasizes non-coherence: executing different programs, different codes, with each getting a different number of vector lanes. They talk about running pure scalar threads for applications that parallelize but which do not vectorize.
 - Much like my NIMT. Although my NIMT has been emphasizing divergence within a group of similar threads, not disparate threads.
 - Emphasizes VL > PVW, with time pipelining. Variable length vectors.
 - "Because vector instructions specify multiple element operations, each instruction occupies a vector functional unit for several cycles even in multi-lane implementations. Hence, the VCL structures are much simpler than their scalar counterparts because a lower instruction issue rate is sufficient in the vector unit (typically one or two instructions per cycle)."
 - Must feed instructions to the vector unit, via scalar logic and/or vector control logic. Issue: replicate or multiplex.
 - The purist Nvidia SIMT has no separation – scalar code runs in the vector lanes.
 - However, I have been thinking of CSEing scalar code common to coherent threads. E.g. loop control.
 - Evaluates on a microarchitecture with scalar/vector. Scalar is superscalar, SMT, OOO.
 - I see absolutely no mention of coherence, of recognizing when the same instruction is being executed by multiple threads. Thus, differs in a fundamental way from the ideas I am exploring..
 - Bottom Line: similar, but substantial differences.

Academic Work: Krashinsky, Asanovic et al, Vector Threading

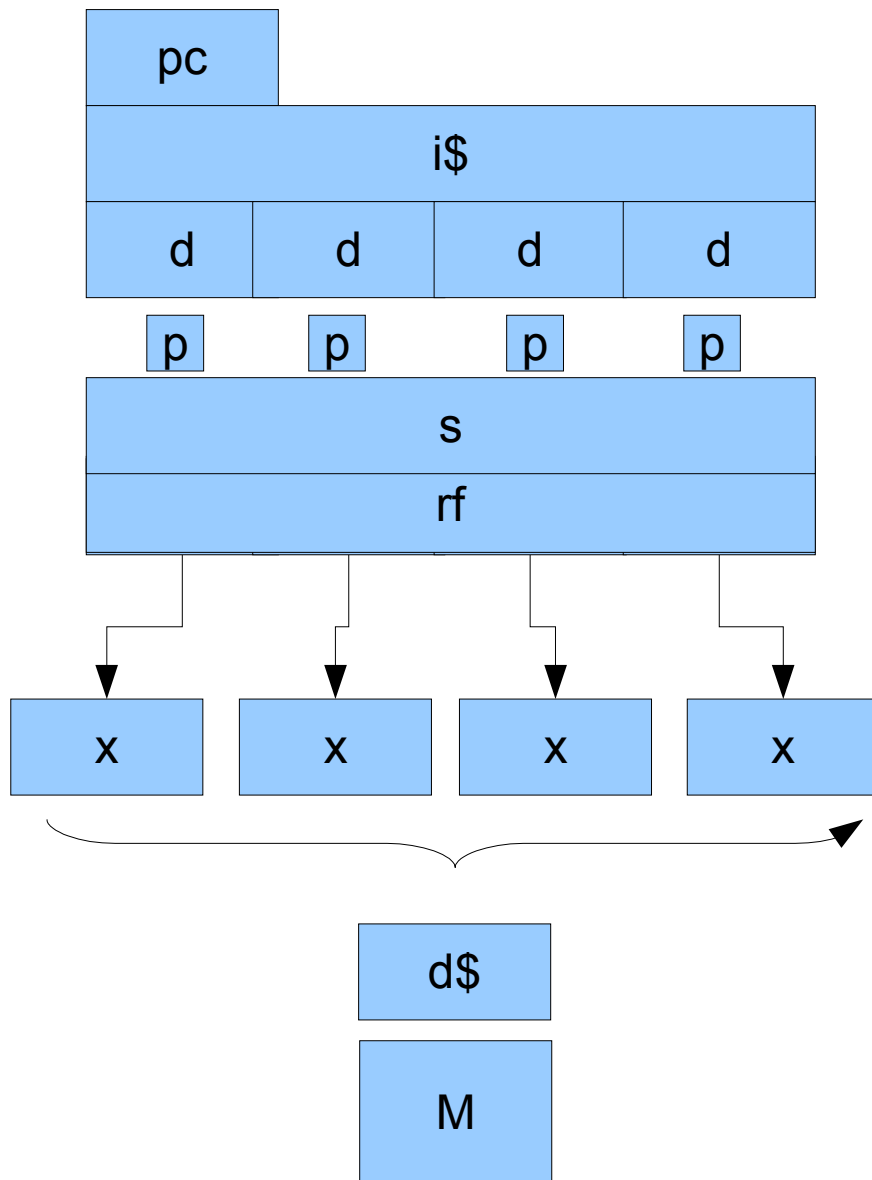
- Krashinsky, Asanovic, et al, Vector Threading (VT). Also, the "Scale" computer architecture
 - Ronny Krashinsky contacted me. I enthusiastically enjoyed his research. This was one of the first "New Vector" papers I encountered.
 - Krashinsky, R., Batten, C., Hampton, M., Gerding, S., Pharris, B., Casper, J., and Asanovic, K. 2004. The Vector-Thread Architecture. IEEE Micro 24, 6 (Nov. 2004), 84-90. DOI= <http://dx.doi.org/10.1109/MM.2004.90>
 - E.g. 4 lanes. VP (vector Processors) striped across lanes. Lanes decoupled from each other.
 - AIBs (Atomic Instruction Blocks), rather like ceddls – no PCs, AIBs request each other, or from control processor.
 - AIBs expose uarch state like chain (bypass) registers at each ALU input.
 - Vector fetch commands are broadcast to all lanes. Each tag checks in an AIB cache. (The moral equivalent of an I\$ per lane; except that the instructions are these AIB packets.)
 - No mention of coherence, sharing of ifetch between separate threads. Seems further away from SIMT.
 - Krste Asanovic recommended the following papers from his group
 - Ronny Krashinsky, Christopher Batten, Mark Hampton, Steven Gerding, Brian Pharris, Jared Casper, and Krste Asanović, "The Vector-Thread Architecture", 31st International Symposium on Computer Architecture (ISCA-31), Munich, Germany, June 2004. <http://www.cag.csail.mit.edu/scale/papers/vta-isca2004.pdf>
 - Mark Hampton and Krste Asanović, "Compiling for Vector-Thread Architectures", International Symposium on Code Generation and Optimization (CGO-2008), Boston, MA, April 2008. <http://www.cag.csail.mit.edu/scale/papers/vtcompiler-cgo2008.pdf>
 - Ronny Krashinsky, "Vector-Thread Architecture and Implementation" Ph.D. Thesis, Massachusetts Institute of Technology, May 2007. <http://www.cag.csail.mit.edu/scale/papers/krashinsky-phd.pdf>
 - Ronny Krashinsky, Christopher Batten, and Krste Asanović, "Implementing the Scale Vector-Thread Processor", ACM Transactions on Design Automation of Electronic Systems (TODAES), 13(3), 41:1-41:24, July 2008. <http://www.eecs.berkeley.edu/~krste/papers/a41-krashinsky.pdf>
 - Christopher Batten, Hidetaka Aoki, and Krste Asanović, "The Case for Malleable Stream Architectures", Workshop on Streaming Systems at 41st International Symposium on Microarchitecture (MICRO-41), Lake Como, Italy, November 2008 <http://www.eecs.berkeley.edu/~krste/papers/maven-micro-wss2008.pdf>

Comparing this Presentation's Coherent Threading (CT) to Krashinsky et al's Vector Threading (VT)

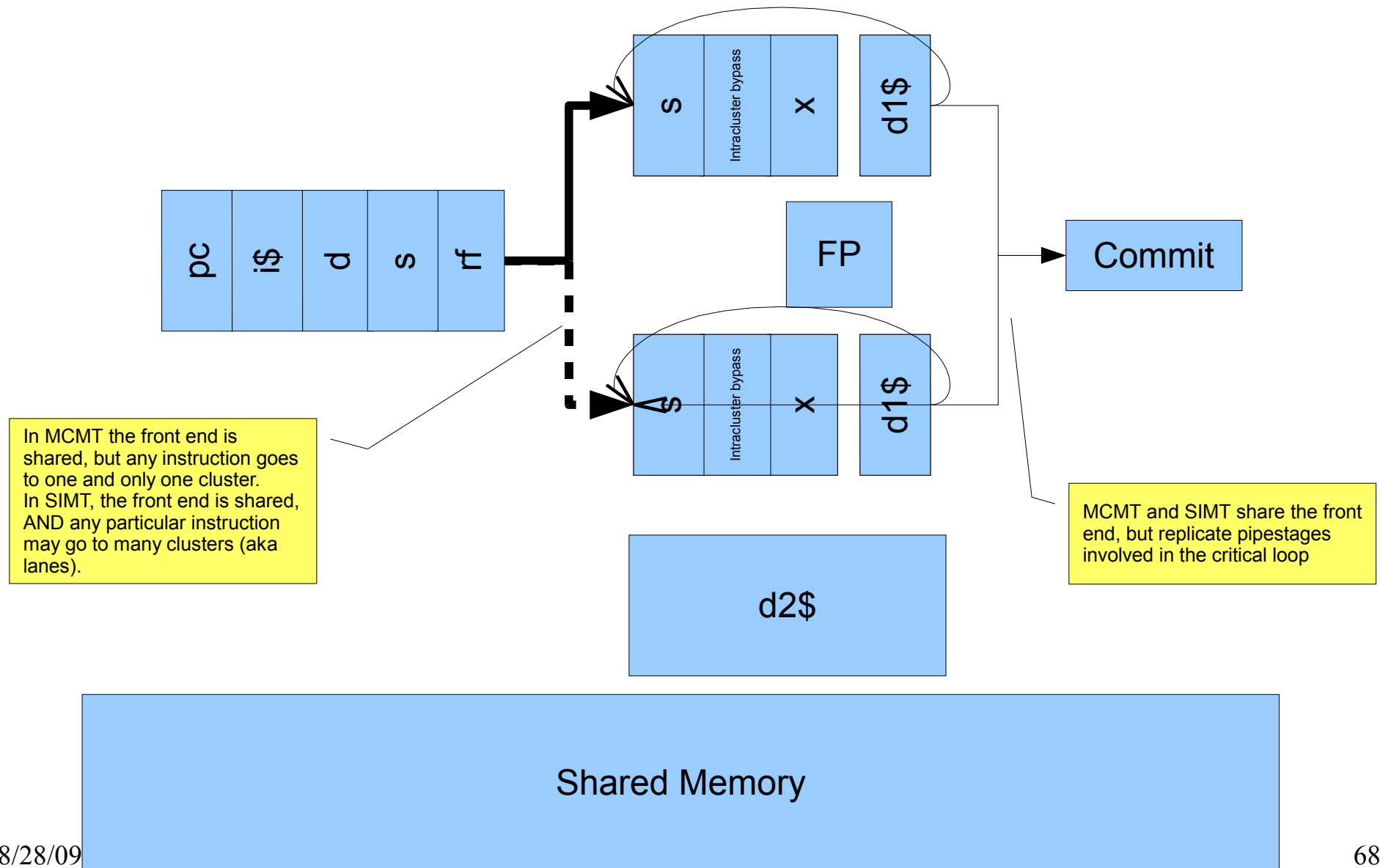
- Many similarities. I admit inspiration by Krashinsky. VT was one of the first examples of “New Vectors” that I encountered. But important differences, to my mind.
- Mainly:
 - VT (SCALE) seems to be a decoupled streaming architecture on a vector-like substrate. SW exposed.
 - CT is a vector-like substrate implementing lane threads. Decoupling, etc., seeking to be hidden, so as not to be an obstacle to MIMD, eventually.
i.e. CT attempts to hide the details as microarchitecture, rather than exposing them to software ISA.

Document

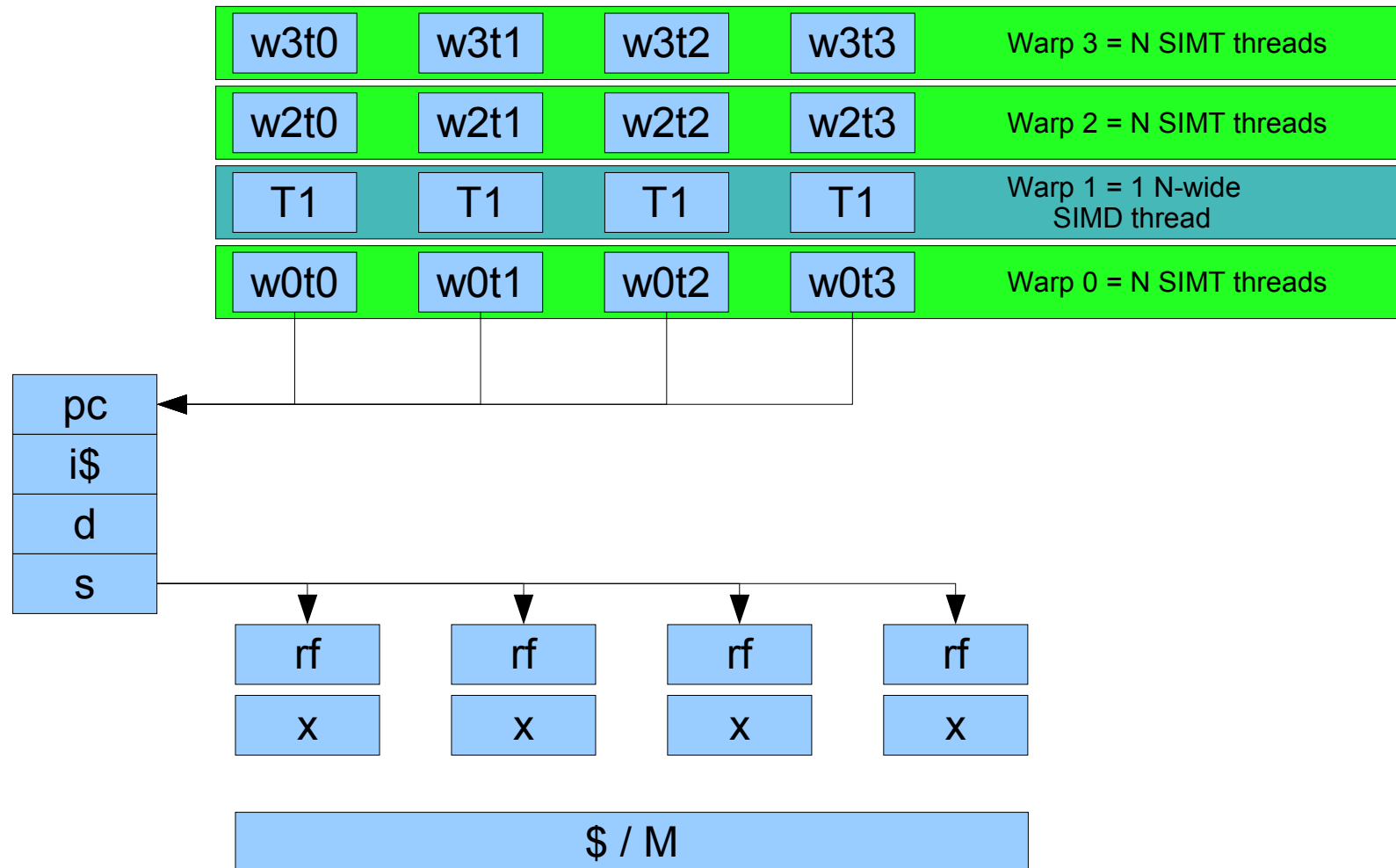
VLIW



Multicluster Multithreading to SIMT



Running SIMD vector parallel code on SIMT vector lane threaded uarch



Running SIMD on SIMT, and vice versa

SIMD on SIMT

- Create a “warp” of N vector lane threads
- Execute in lockstep
(which happens naturally on SIMT)
- Use predication or branches to emulate vector masks

Why?

To speed up latency sensitive thread? (even if it wastes ...)

Which do you think is more efficient?

SIMT on SIMD

Replace all masked operations

$$Vd = \text{op}(Va, Vb) \text{ under } K$$

With tests for null mask

```
IF K = 0 goto Skip
Vd = op(Va, Vb) under K
Skip: ...
```

If null masks are at all likely.

Why?

Because it can be faster even on a SIMD;
better still to reorganize code to branch around
many such operations.

But doesn't take advantage of SIMT HW
optimizations to improve utilization.

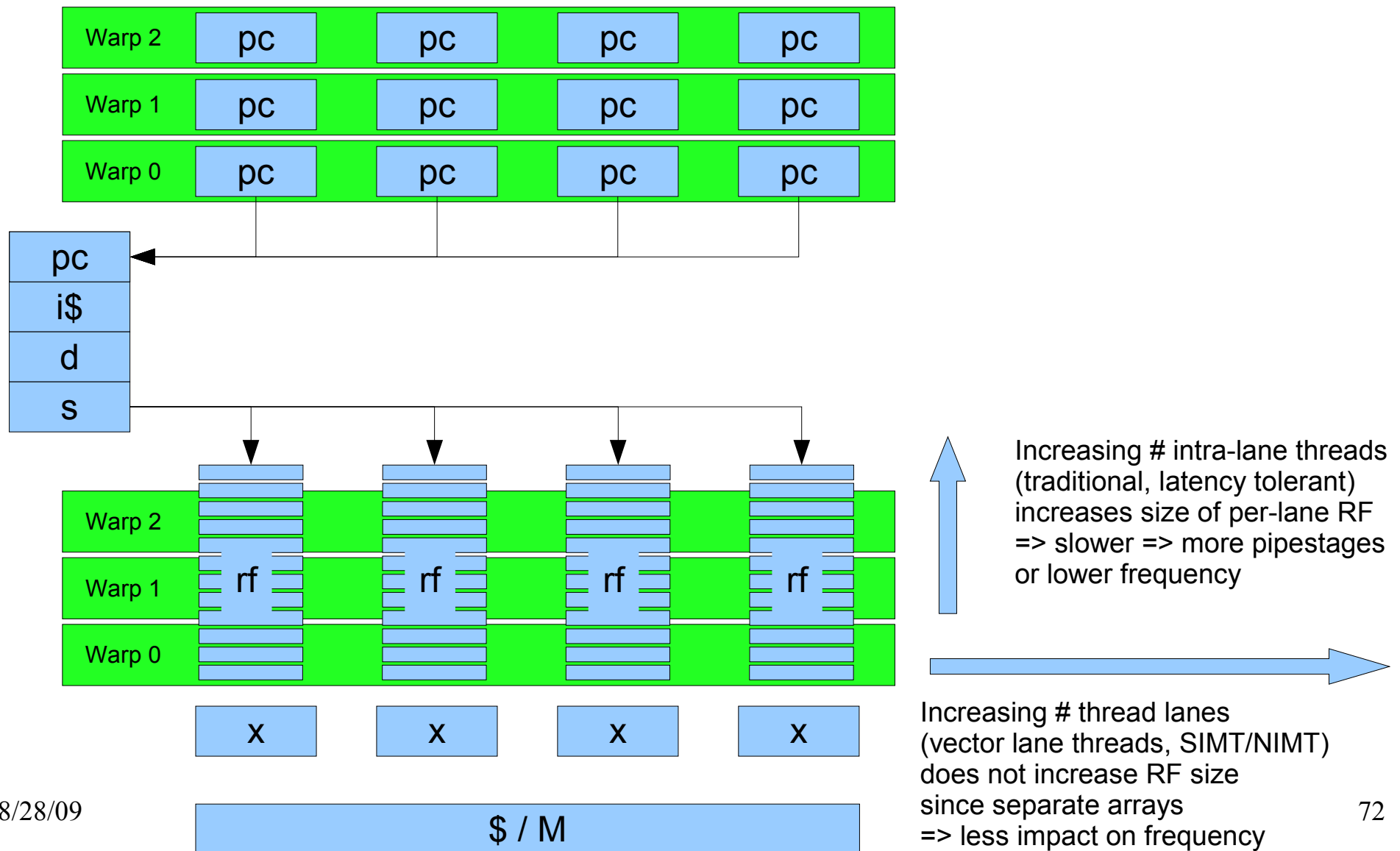
Data Pattern Dependent Power Optimization Time Pipelined Vectors

- Observation: adjacent pixels close in value

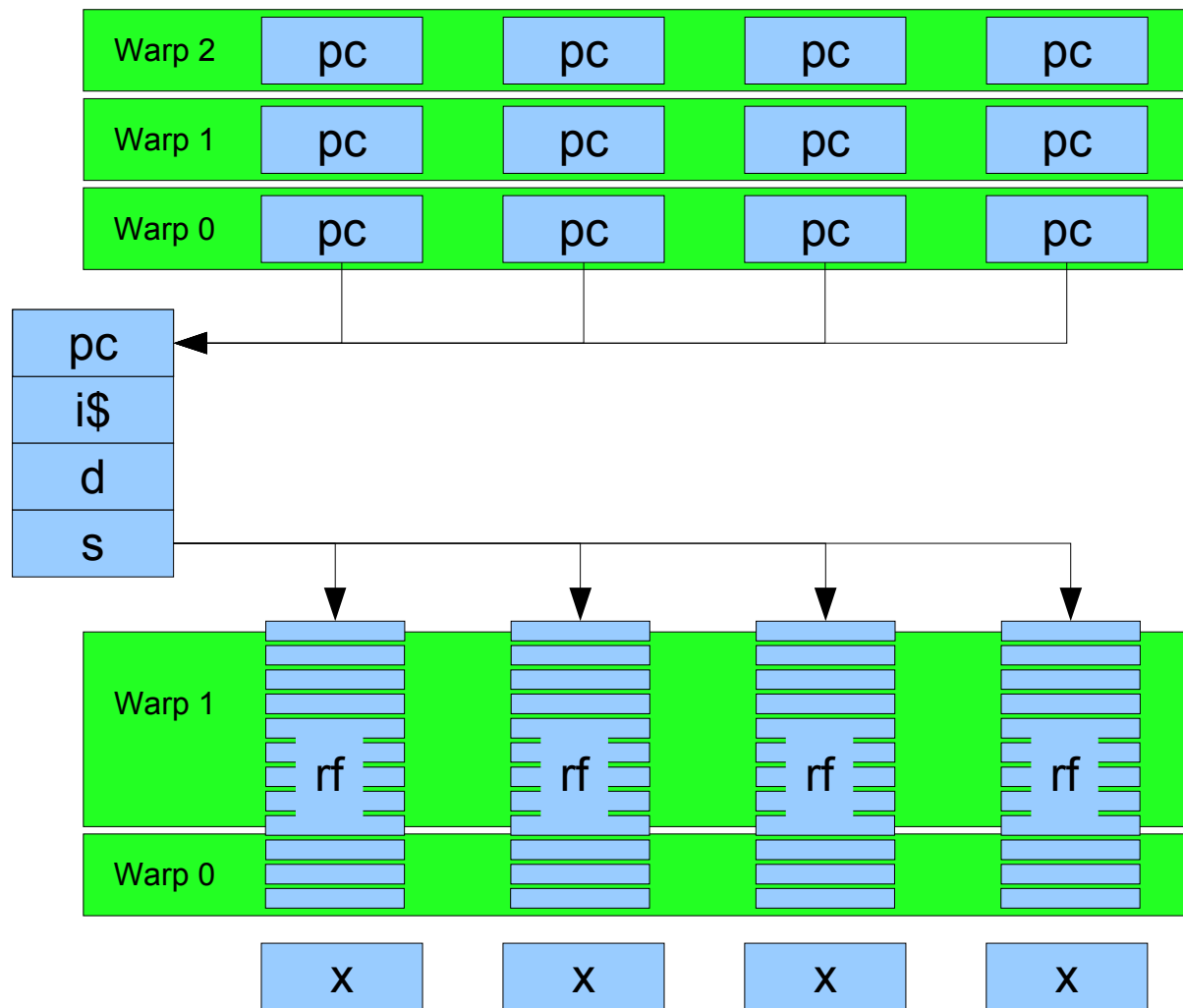
Planar (soa) or Chunky (aos: xyzw or rgba or ...)

- SIMD vectors put adjacent pixels in different vector lanes. This is bad.
- Running nearby pixels through same vector lane
⇒ less toggling ⇒ less power
- Credit: TP
 - Idea: instead of 16 wide SIMD, implement 4 sets of 4 wide SIMD taking 4 cycles to execute a 16 wide vector instruction. With chaining. (Complicated control.)
 - Or: use SIMT, with vector lane threads operating on chunks (e.g. cache lines), as opposed to partitioning a cache line between vector lane threads. (Same optimization as for MIMD.)

Intra-lane vs Inter-lane Register Files



Variable Size Register Files



All (?) GPUs have variable numbers of registers per thread (or per SIMT thread group, e.g. warp).

⇒ Thread Id selects PC + RF.base
RF.base + register number distributed

The diagram illustrates the execution of a warp on a GPU. The top section shows a warp of 8 threads (4 in Warp 0, 4 in Warp 1) at time t_0 , each with a program counter (pc). The bottom section shows the warp after a branch, with threads grouped into four sets of four. The top set of four threads (Warp 0) has reached the branch point and is now in a 'rf' (ready for fork) state, while the bottom set of four threads (Warp 1) is still at the start of the branch.

E.g. Nvidia warp = 4X
larger than number of
lanes

E.g. ATI wavefront =
4X larger than number
of (vliw) lanes

```
Preg# = thread.rf.base
+ Lreg# *4 + cycle#
```

SIMT register lookup: simple vs. dynamic

- Simple lookup

$$\begin{aligned}\text{Preg\#} &= \text{thread.rf.base} + \text{Lreg\#} * 4 + \text{cycle} \\ &= \text{rf.base}[\text{thread\#}] + \text{Lreg\#} * 4 + \text{cycle}\end{aligned}$$

Lreg# must be distributed to vector lanes with each instruction. Other values can be preloaded.

- More dynamic lookup

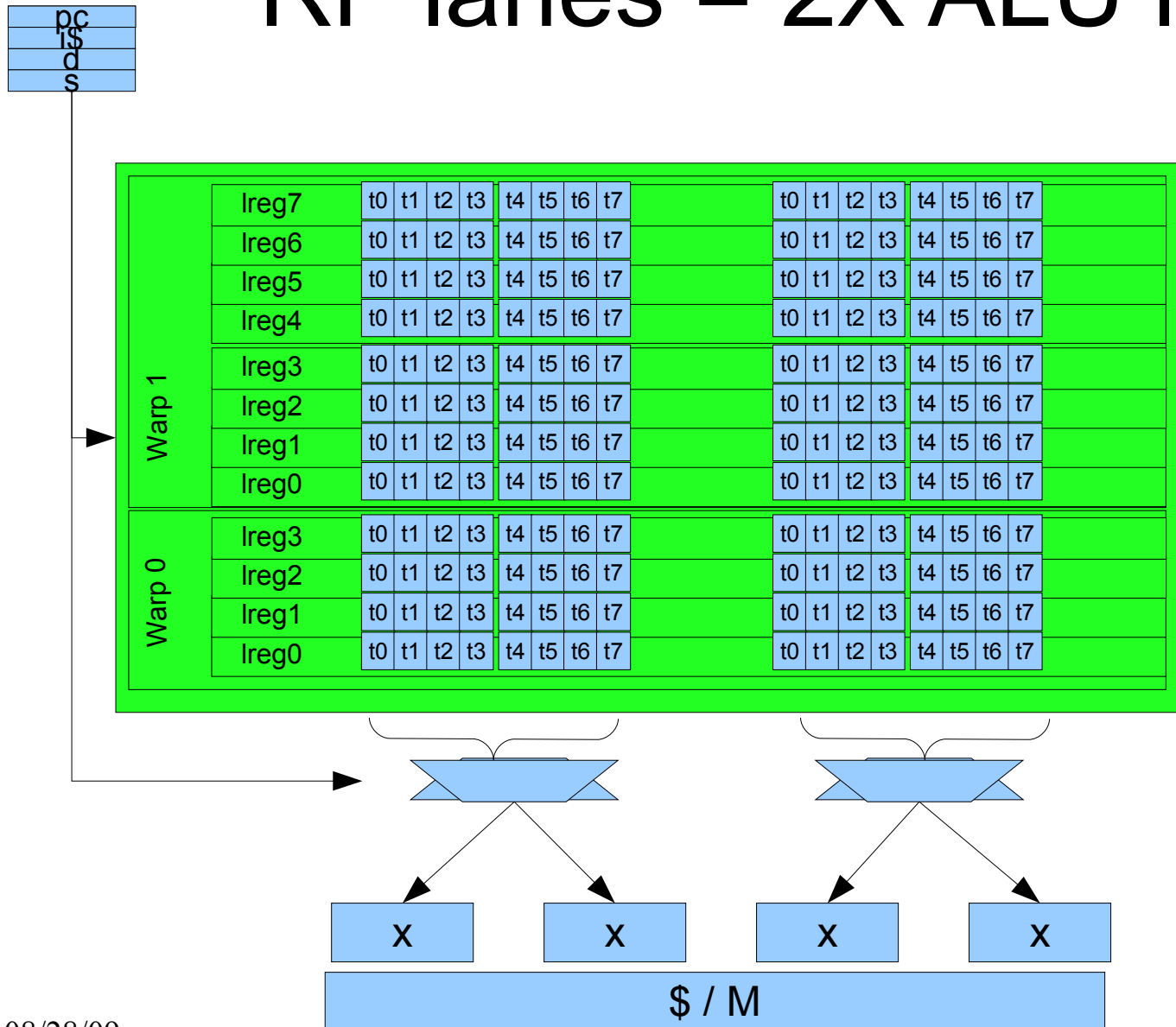
$$\text{Preg\#} = \text{thread.rf.base} + \text{Lreg\#} * 4 + \text{thread_in_lane}$$

i.e. 2 more bits per lane per instruction must be distributed.

Or: compute from lane mask that controls whether thread is active for a particular NIMT instruction

SIMT RF: trading width for ports

RF lanes = 2X ALU lanes



If each RF “lane” maps to 2 ALU “lanes” then rejiggering has still more opportunities to improve VPU utilization.

Reduce RF ports by accessing multiple cycles' worth of registers in one go.

RF muxing may already exist for scatter gather and strided register accesses.

(Some GPUs have s/g RFs)

The GP RF problem

- Consider
 - ISA with fixed #regs/thread
 - e.g. 4 time or latency threads
 - 16 lanes
 - $R \times W$ regs ($W=16 \times 32=512$ bits)
- SIMD parallel has
 - $4R \times W$ regs total
- Lane threading has logical RF per lane per time/latency thread
 - 16 lanes: $16 \times \text{\#regs}$
 - Worse, only reading 32 bits per lane per cycle = $16 \times$ “higher”
 - Motivates wider thread lanes
 - Not 32 bits
 - Perhaps 128 bits? 4×32
 - Still not good
 - $4 \times \text{\#regs}$, $4 \times$ “higher”

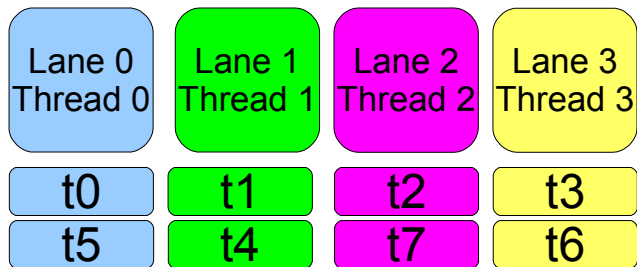
Most threads don't use all regs.

Want variable #regs/thread. Hard in fixed ISA.
? 2 level RF microarchitecture? RF \$?

Change from lane threads to SIMD threads as reg usage grows?

Rejiggering Threads between Cycles of a multi-cycle Warp

Original, Imbalanced = 50% utilized

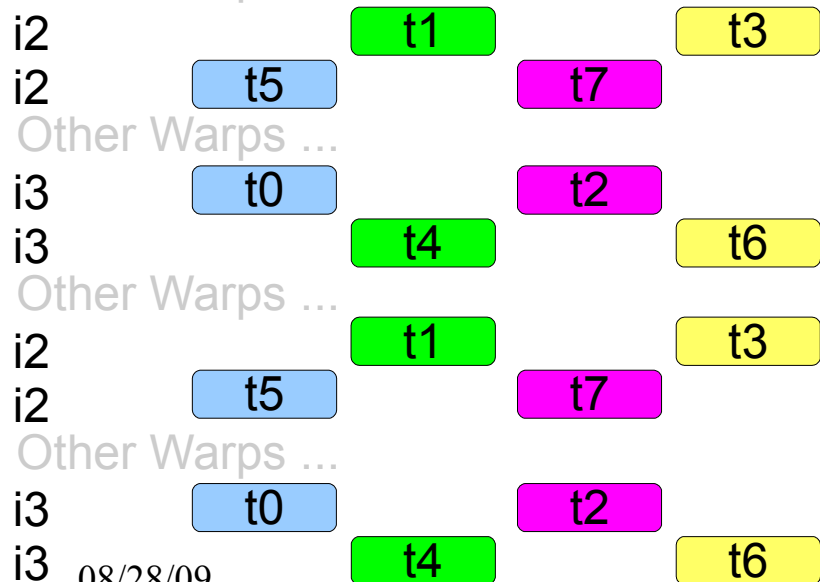


Example assumes
Thread skewing
between waves
in a wavefront

```

I1
repeat i=0 to ...
  for threads t=0 to 7
    if even(i) XOR even (t)
      then I2
    else I3
I4
    
```

Other Warps ...



08/28/09

Rejiggered within Wavefront = 100% utilized



Other Warps ...

At least 4 levels of thread/warp rebalancing

- Rejiggering within a warp (wavefront, multicycle SIMT thread group) between cycles (phases, waves) of same warp
 - Fairly easy – just a few more bits / logic on mask
 - Migrating (exchanging) threads between warps but staying in same lane
 - Fairly easy – more bits
 - Migrating threads between lanes of same warp
 - Hard – involves copying registers.
 - Probably not worth doing, except for RF lane tricks
 - Rebuilding warps (wavefront, multicycle SIMT thread group) completely
 - Hard – involves copying registers
- 08/28/09 – Probably only worthwhile after really long latency events.

Summary of SIMT VPU Utilization Improvements

- SIMT inherently and fundamentally is more efficient, filtering out null masks. (Minor)
- SIMT + vector time pipelining
 - More efficient masking and VL (Major)
 - Extra instruction fetch opportunities during multicycle vector operations. (Major)
 - Also: extra instruction fetch opportunities during multicycle wavefronts (Major)
- SIMT rebalancing
 - Rejiggering within warp: same lane, different cycles (Major)
 - Migrating between warps at same lane (Minor)
 - Rebuilding warps (Major, but complex)

Are SIMT threads Latency Tolerant threads?

Yes and No

- No
 - GPUs always (?) have classic latency tolerant threads – switch to thread that uses some Eus to tolerate latencies, both EU and memory latency.
 - SIMT threads are not used to tolerate EU latency
 - Yes:
 - SIMT threads generate more cache misses, more MLP
 - In the absence of time multiplexed threads, they just idle their share of the vector lane when so doing.
 - Similar to MIMD
 - Q: Do non-threaded MIMD cores tolerate latency?
 - A: yes: they generate MLP, but idle cores
 - MIMD is MLP for when cores are cheap
- 08/28/09 SIMT is just a step on the way to MIMT – for when cores, and x86 decode, etc. are not so cheap 81

Are SIMT threads really threads at all?

- GPUs also have classic latency tolerant threads
 - These form “groups” of the SIMT coherent vector lane threads.
- Different companies have different terminology

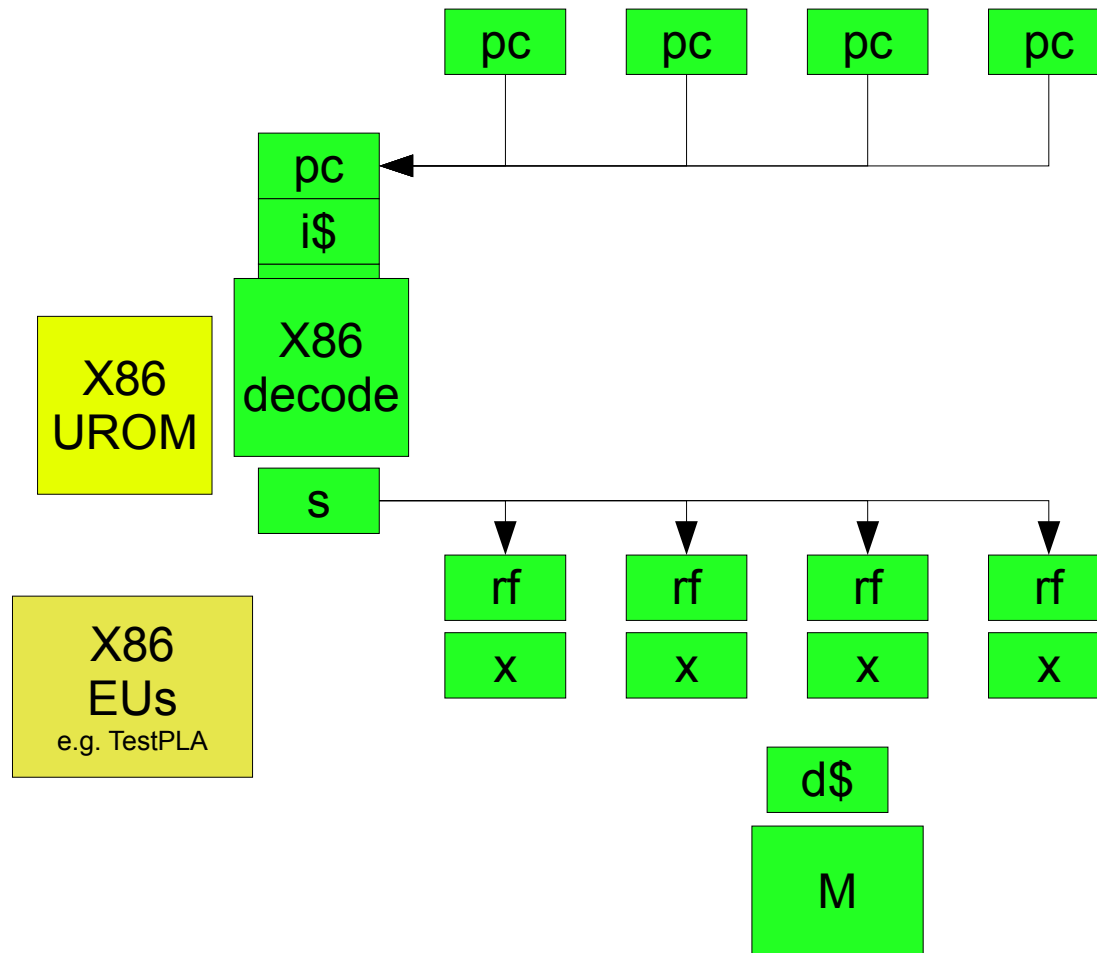
	<u>Multilane Groups</u>	<u>Vector Lane “Threads”</u>
Nvidia	Warps	Threads
ATI	Wavefronts	?
Intel GenX	Threads	? channels ?

Some say: calling them threads is just dishonest marketing terminology

Me: they fill the software definition of threads (PC, SP, regs)

- So I asked the original inventors of Fine Grain Multithreading
 - Arvind (MIT)
 - Burton Smith (MIT, HEP, Tera)
 - They say: Yes, they are threads...
 - Just not exactly the same, not as capable, as “horizontal” latency tolerant threads
- Coherent Vector Lane Threads

X86 on a Vector Lane Threaded SIMT Microarchitecture



No problem!

Unlike many microarchitecture ideas, x86 works well with SIMT.

X86 overhead – decoder, UROM – is naturally amortized over multiple SIMT vector lane threads.

Even x86 execution units like TestPLA can be shared.

No need for binary translation. But, of course, can use BT, e.g. to do speculation & optimization.

ISA extensions for SIMT – *N / Y*

- None are necessary
 - i.e. there is no need for ISA extensions specifically for SIMT
 - If we have hardware or hints to detect control independence
 - Apart from usual CUID, etc.
- None are desirable
 - No ISA extensions should be added for SIMT which are not also desirable for MIMD.
 - SIMT is just a step on the way to MIMD.
- Nevertheless...
 - Control Independence Hints
 - IF ELSIF ELSE ENDIF
 - LOOP BREAK ENDLOOP
 - CUID bits for stride
- Generic Good Stuff
 - Variable #lregs / thread
 - Register kill hints
 - VL / PVW variable length vectors
 - Lightweight Threading
 - Barriers, other Synchronization
 - Cache Control/Bypass Directives
 - Message Passing

SIMT Vector Lane Threading and Enterprise-level Reliability

0.999999... - No problem!

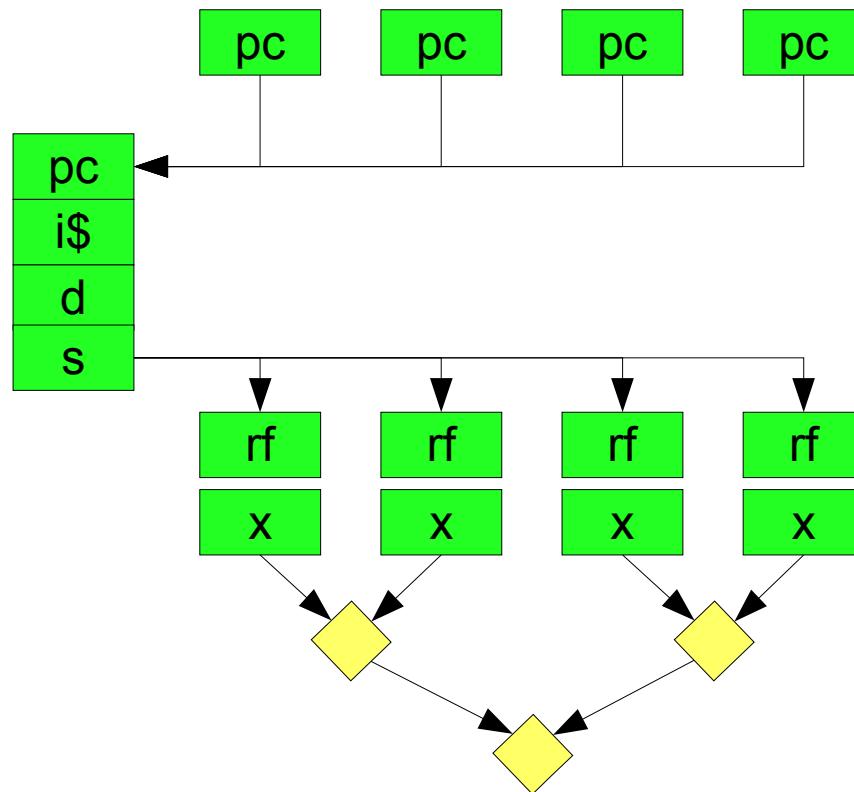
Remember lockstep?
FRC? QMR?

Remember how we had to abandon
lockstep at chip, and even core,
boundaries?

SIMT vector lane threads naturally
run in lockstep. Add checkers...

Better than RMT: spatial diversity.
(add delay for temporal diversity)

Front end shared between threads:
DIMIT.



What's this guff about “Coherence”?

- “Coherence”
 - standard term in GPU design
 - “Coherent” as in LASER light
- Instruction Coherence
 - Multiple vector lane threads executing same instruction PC at same time
 - ⇒ can share ifetch
 - Much discussed in this presentation.
- Data Coherence
 - Multiple vector lane threads executing same, or similar, data memory references at same time
 - ⇒ maybe share memory ports?

Data Coherence

- Design challenge / opportunity
- Different lanes fetching
 - exactly same data
⇒ trivial sharing
 - strided data in same/different cache line(s)
⇒ less trivial. Share decode? TLB? Fill buffers?
 - Scatter / gather ... packetization
- If threads are control flow diverged
are data memory addresses diverged?
- If threads are rearranged,
is data coherence lost?
 - Can it be regained?
- Types of data coherence
 - Exactly same memory location in different lane threads.
 - Adjacent (stride 1) memory locations
 - Stride-N
 - Scatter/gather
 - Small LUTs
 - Widely separated
 - Different memory addresses, same value
 - e.g. different threads, local variable, spilled loop counter, same value

Intra vs. Inter Cacheline Threading

- SIMD vectors work best dense, stride-1
 - Successive vector elements adjacent in cache line
- Intra:
 - Neighbouring SIMT threads work on adjacent elements in cache line
 - Or... elements in same cache line, but not necessarily adjacent. (Intra cache line scatter/gather.)
- Inter: SIMT threads work on entire cache line
 - e.g. using time pipelined vector instructions
 - e.g. neighboring SIMT threads work on neighbouring cache lines
- MIMD prefers inter cacheline threading
- SIMT prefers intra; inter \Rightarrow larger cache footprint
- Possibility: CPUID info, like VL/PVW strip mining

Why (Coherent Vector Lane) Threads are friendlier to software than SIMD vector parallel

- I found this strange
 - As an old vector guy (Gould), I expected vectors to be well understood and liked
- However
 - Supercomputer guys prefer MIMD-scalar
 - most flexible,
 - only one degree of optimization
 - Microsoft DX and OpenGL are SIMT
 - Not vector – compiler must convert SIMT threads to vector instructions, i.e. must unparallelize
 - May be chicken and egg... but there you are.
- Robust wrt Change
 - Fixed width SIMD vectors must be recoded when width changes, e.g. 512 → 1024b
 - Short vectors are inefficient on wide vector architectures
 - SIMT lanes individually narrower vectors
 - SIMT code can run on MIMD, or on wider SIMT (esp. using techniques here)

Strawman Microarchitecture

To match a 512-bit wide SIMD in throughput
and beat it in utilization and SW ease

- Rule of 4
 - 512-bit wide datapath
 - 4 “lanes” \Rightarrow 4 vector lane threads
 - Each lane 128 bits wide = 4×32
 - SIMT (or maybe DIMT)
 - 4 cycles (or waves) per warp (or wavefront)
 \Rightarrow 16 threads per warp
 - But remember using the tricks here
 \Rightarrow effectively 2 threads / warp comparable
 - Warps need not take 4 cycles, can be as little as 1.
 - 4 warps (i.e. 4 “true threads”)
 - Cache: TBD