# 18-640 Foundations of Computer Architecture

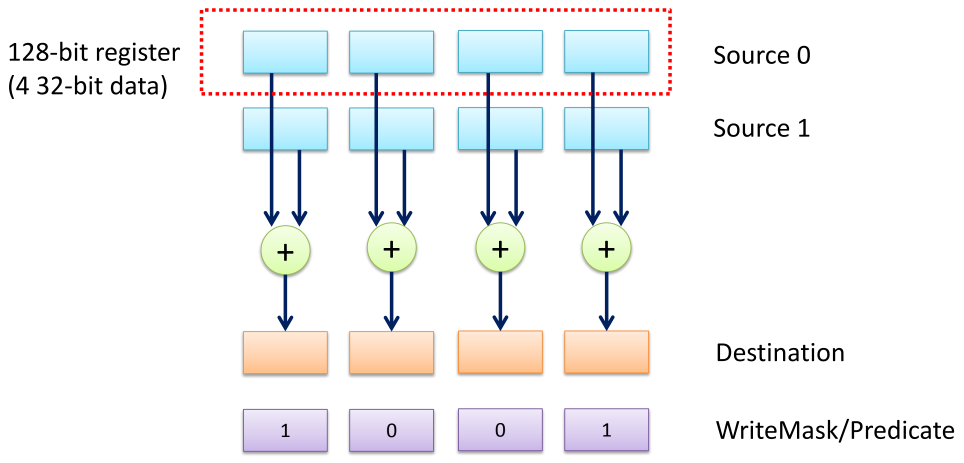
## Lecture 16:
## "SIMD and GPU Processors"

Mridula Chappalli Srinivasa
(Ken Lueh, PhD - Sr. Principal Engineer, Intel)
October 30, 2014

- Required Reading Assignments:
  - Chapter 4 from new Hennessy & Patterson textbook
    - Computer Architecture, 5th edition
    - Read 4.4 – 4.10 for today's lecture

Electrical & Computer ENGINEERING

## Single Instruction Multiple Data (SIMD)



## SIMD Extensions for Superscalar Processors

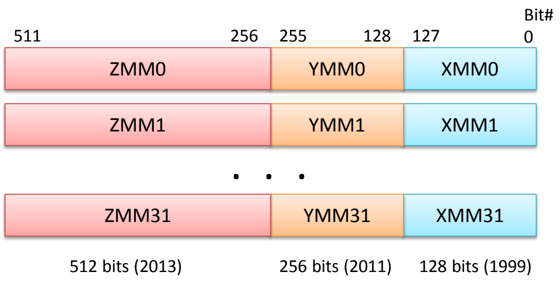
- Every CISC/RISC processor today has SIMD extensions
  - MMX, SSE, SSE-2, SSE-3, SSE-4, AVX, AVX2, Altivec, VIS, …
- Basic idea: accelerate multimedia processing
  - Define vectors of 8, 16, 32 and 64 bit elements in regular registers
  - Apply SIMD arithmetic on these vectors
- Nice and cheap
  - Don't need to define big vector register file
    - This has changed in more recent SIMD extensions
  - All we need to do
    - Add the proper opcodes for SIMD arithmetic
    - Modify datapaths to execute SIMD arithmetic
  - Certain operations are easier on short vectors
    - Reductions, random permutations

## Problems with SIMD Extension

- SIMD defines short, fixed-sized, vectors
  - Cannot capture data parallelism wider than 64 bits
    - MMX (1996) has 64-bit register s (8  8-bit or 4 16-bit operations)
  - Must use wide-issue to utilize more than 64-bit datapaths
  - SSE and Altivec have switched to 128-bits because of this
  - AVX2 has switched to 512-bits because of this
- SIMD does not support vector memory accesses
  - Strided and indexed accesses for narrow elements
  - Needs multi-instruction sequence to emulate
    - Pack, unpack, shift, rotate, merge, etc
  - Cancels most of performance and code density benefits of vectors
- Compiler support for SIMD?
  - Auto vectorization is hard
  - Reply on programming model (e.g., OpenMP, Cilk+)

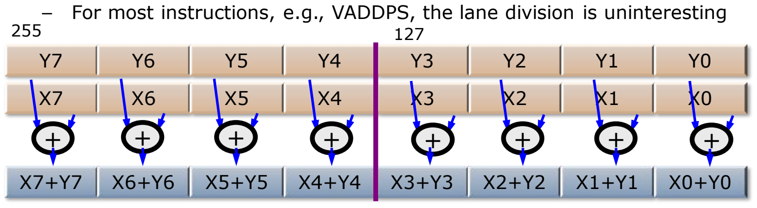
## AVX2 SIMD Register Set

- Intel® AVX extends all 16 XMM registers to 256bits
- Intel AVX instructions operate on either:
  - The whole 256-bits (FP only)
  - The lower 128-bits (like existing Intel® SSE instructions)
    - A replacement for existing scalar/128-bit SSE instructions
      - Provides new capabilities on existing instructions
    - The upper 128-bits of the register are zeroed out
- Intel AVX2 supports integer operations

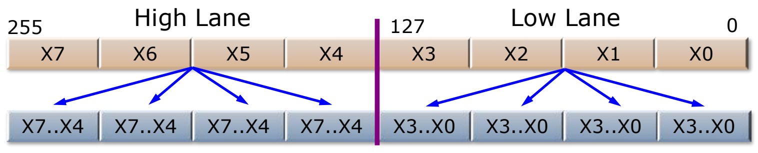


## AVX Example

- Intel® AVX defines two 128-bit lanes (low =xmm, high=ymm[255:128])
  - Nearly all operations are defined as "in-lane"
  - For most instructions, e.g., VADDPS, the lane division is uninteresting



- Some in-lane behavior is more interesting: VPERMILPS



***Intel® SSE functionality is preserved within lanes***

## Graphics Processors Timeline

- Till mid 90s
  - VGA controllers used to accelerate some display functions
- Mid 90s to mid 00s
  - Fixed-function graphics accelerators for the OpenGL and DirectX APIs
    - Some GP-GPU capabilities by on top of the interfaces
  - 3D graphics: triangle setup & rasterization, texture mapping & shading
- Modern GPUs
  - Programmable multiprocessors (optimized for data-parallel ops)
    - OpenGL, DirectX, C++ AMP, OpenCL, RenderScript, Cilk+
  - Some fixed function hardware (texture, raster ops, …)

## Why GPU?

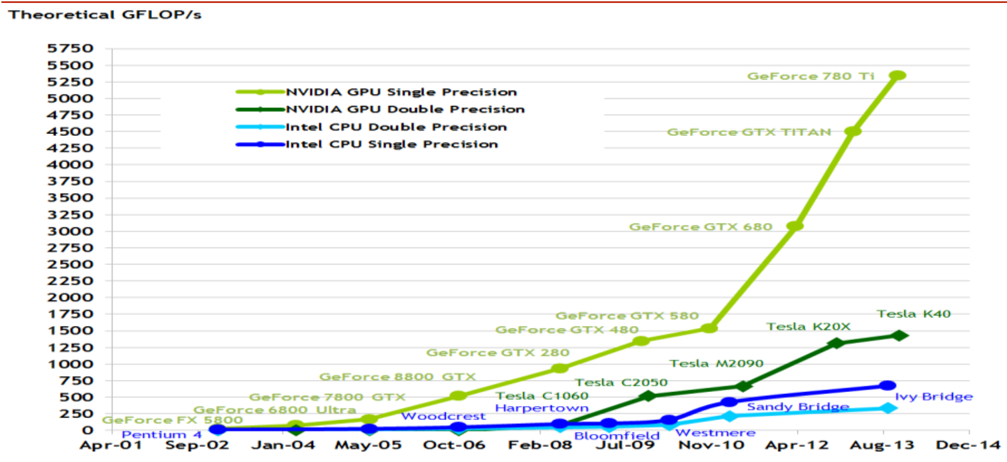


Fig 2: NVIDIA CUDA C Programming Guide

## CPU vs GPU

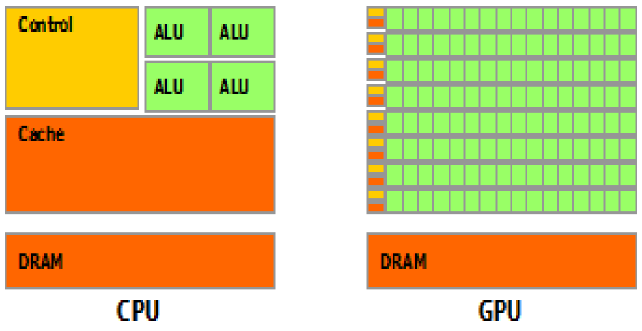
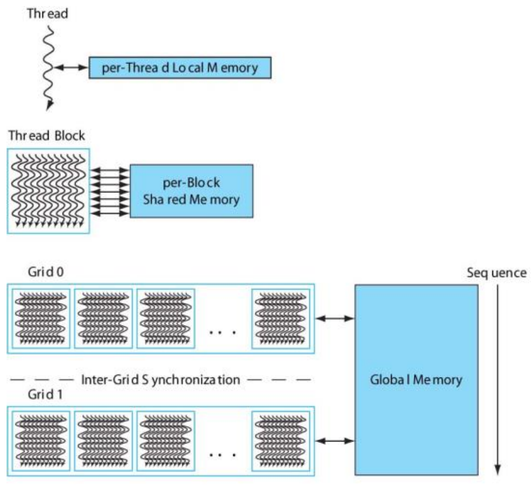


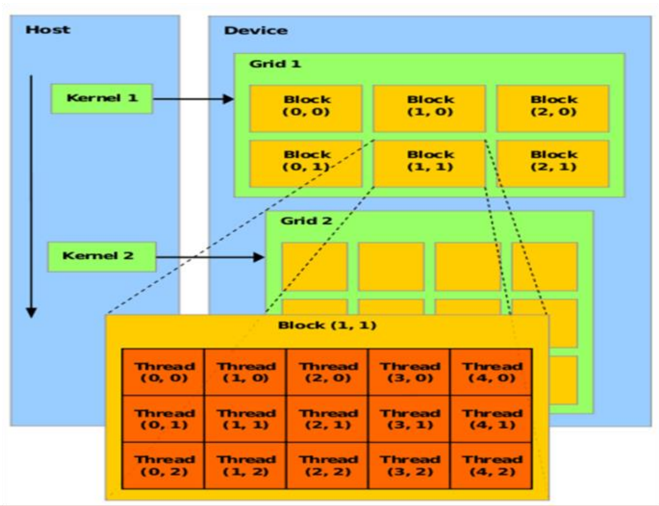
Fig 3: NVIDIA CUDA Programming Guide

## GPU Thread Model - Software View



- Single-program multiple data (SPMD) model
- Each thread has local memory
- Parallel threads packed in blocks
  - Access to per-block shared memory
  - Can synchronize with barrier
- Grids include independent groups
  - May execute concurrently

## Data Parallel Decomposition



- Programmer partitions problems into grids and blocks
- Programmer partitions blocks into threads
- Built-in variables
  - blockIdx.x, blockIdx.y, blockIdx.z
  - threadIdx.x, threadIdx.y, threadIdx.z
  - blockDim.x, blockDim.y, blockDim.z
  - gridDim.x, gridDim.y, gridDim.z

## Code Example: DAXPY

C Code

```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
            for (int i = 0; i < n; ++i)
                    y[i] = a*x[i] + y[i];
}
```

CUDA Code

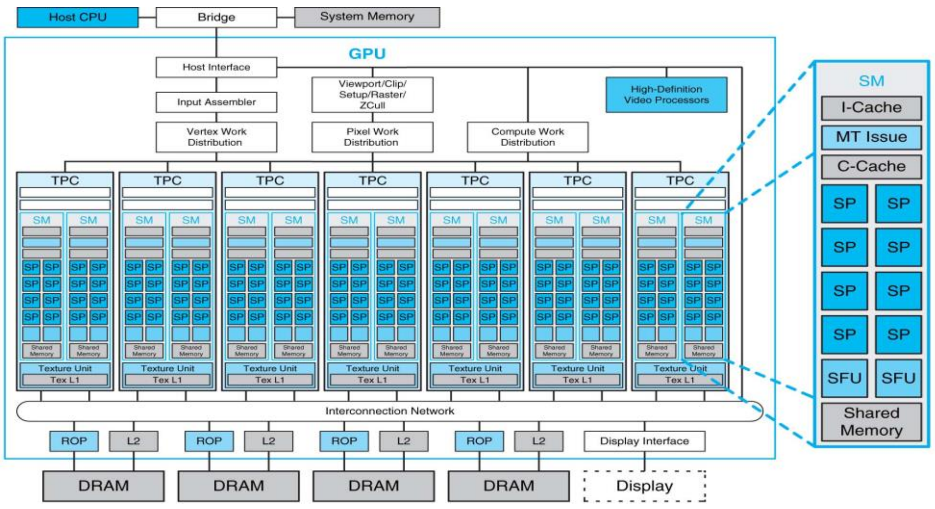
```
// Invoke DAXPY with 256 threads per block
__host__
int nblocks = (n+ 255) / 256;
    daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

- CUDA code launches 256 threads per block
  - Thread = 1 iteration of scalar loop (1 element in vector loop)
  - Block = body of vectorized loop (with VL=256 in this example)
  - Grid = vectorizable loop

## Sample CUDA Program

```
__global__ void vcos( int n, float* x, float* y ) {
        int ix = blockIdx.x*blockDim.x + threadIdx.x;     }
int main() {
   float *host_x, *host_y;
   float *dev_x, *dev_y;
   int n = 1024;
   host_x = (float*)malloc( n*sizeof(float) );
   host_y = (float*)malloc( n*sizeof(float) );
   cudaMalloc( &dev_x, n*sizeof(float) );
   cudaMalloc( &dev_y, n*sizeof(float) );
  /*TODO – Fill x[i] with data */
  cudaMemcpy( dev_x, host_x, n*sizeof(float), cudaMemcpyHostToDevice );
   bk = (int)( n / 256 );
   vcos<<<bk,256>>>( n, dev_x, dev_y );
  cudaMemcpy( host_y, dev_y, n*sizeof(float), cudaMemcpyDeviceToHost );
  return(0) }
```

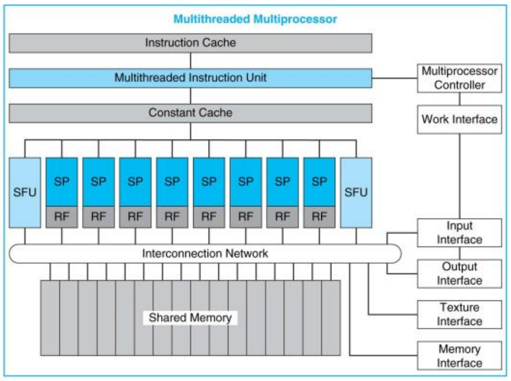
## GPU Architecture: Nvidia GeForce 8800 (aka Tesla Architecture)



## GPU Architecture

- A highly multithreaded, multiprocessor system
  - 100s of streaming processors (SPs)
  - 8 SPs in a streaming multiprocessor (SM) with some caches
  - 2 SMs in a texture processor cluster (TPCs) with one texture pipe
- Scheduling controlled mostly by hardware
- Scalability
  - By scaling the number of TPCs and memory channels
- Fixed function components for graphics
  - Texture pipes and caches, raster operation units (ROP), …

## Streaming Multiprocessor

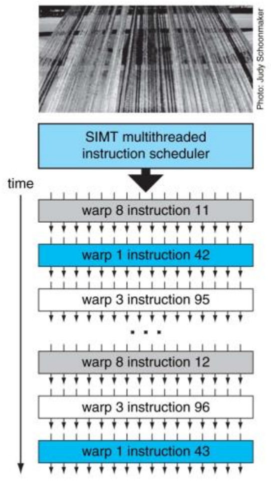


- Another way to think of this
  - SM = a multithreaded vector core
  - SP (streaming processor) = a vector or SIMD lane (registers + ALUs)

## Streaming Multiprocessor Details

- Each SP is a simple processor core
  - 1024 32-bit registers shared flexibly by up to 64 threads
  - Integer and floating-point arithmetic units
    - Including multiply add
- Special function unit
  - Implements functions such as divide, square root, sine, cosine, …
- Instruction cache and constants cache
  - Shared by all threads
- Multibanked shared memory
  - E.g. 16 banks to allow parallel accesses by all SP

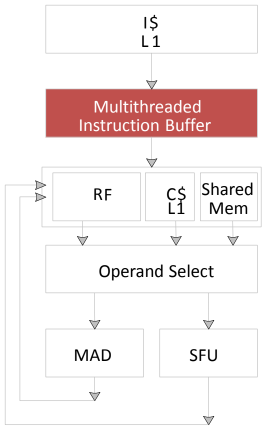
## Instruction & Thread Scheduling: Where Threads Meets Data Parallelism



- In theory, all threads can be independent
  - SM hardware implements zero-overhead switching
  - 768 threads (24 warps)
- For efficiency, 32 threads are packed in warps
  - Warp: set of parallel threads the execute same instruction
    - Warp = a thread of vector instructions
    - Warps introduce data parallelism
  - 1 warp instruction keeps SPs busy for 4 cycles in Tesla
- Individual threads may be inactive
  - Because they branched differently or predication
  - This is the equivalent of conditional execution
  - Loss of efficiency if not data parallel
- Software thread blocks mapped to warps
  - When HW resources are available

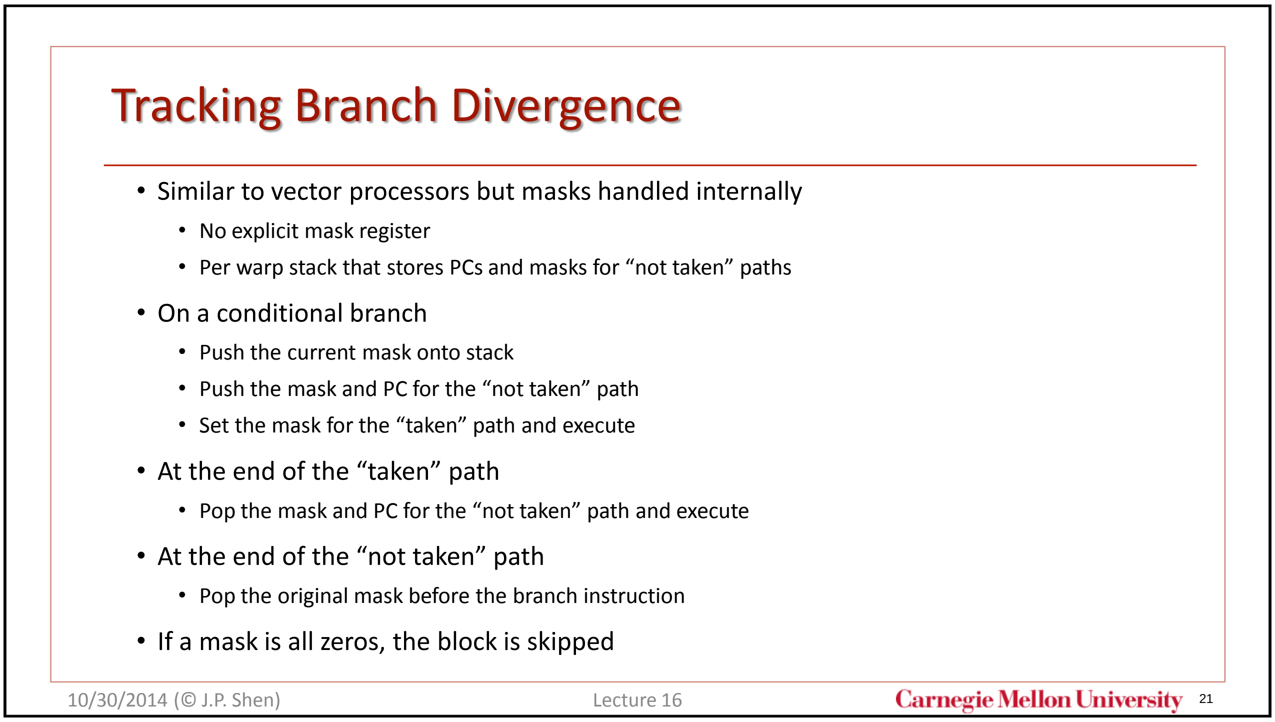
## Instruction Buffering & Warp Scheduling

- Fetch one instruction/cycle
  - From the L1 instruction cache into an instruction buffer slot
- Issue one "ready-to-go" instruction/cycle
  - All elements of the warp must be ready
  - Scoreboarding used to track hazards and determine ready warps
  - Round-robin or age based selection between ready warps
- Instruction broadcasted to all SP
  - Will keep SPs busy for up to 4 cycles



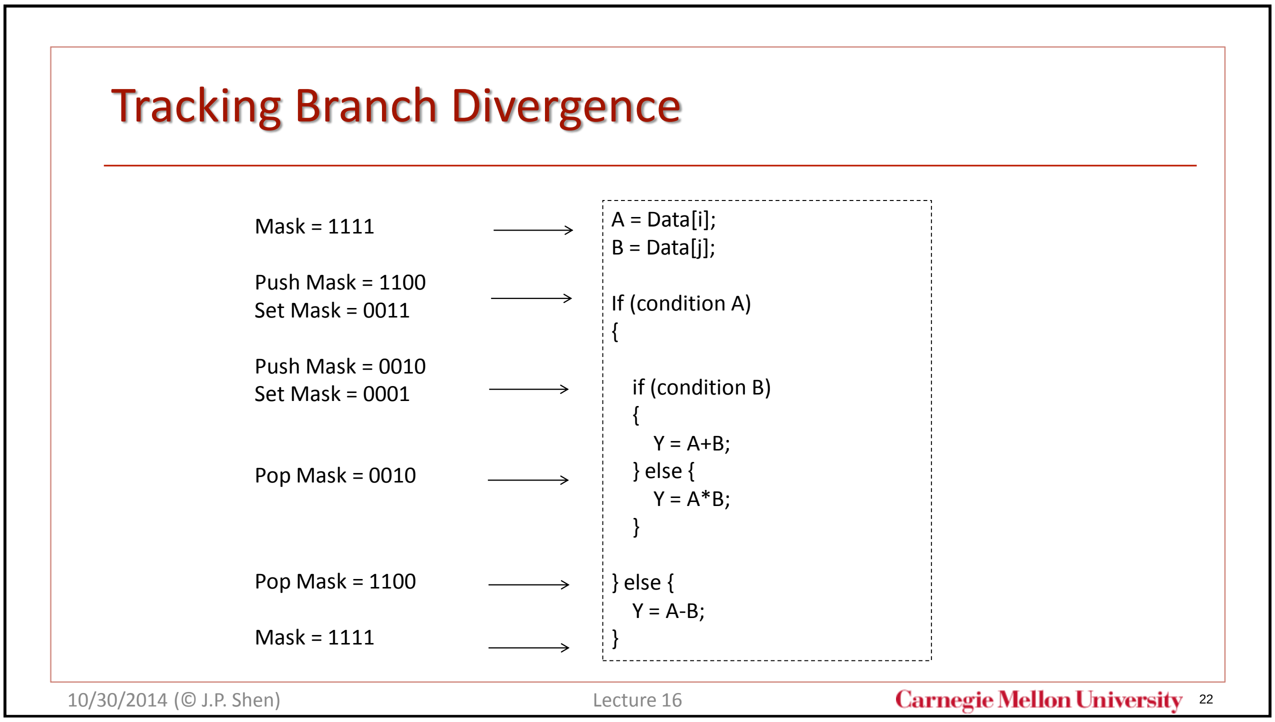
## Dependence Tracking Using a Scoreboard

- Status of all register operands is tracked
  - RAW hazards for high-latency operations
  - Dependencies to memory operations
- Instructions are ready when all register operands are ready for whole warp
  - Divergence of threads within warps is also tracked
- A warp may be blocked because of
  - Dependencies through registers
  - Synchronization operations (barriers or atomic ops)
  - Memory accesses
- But other warps can proceed in order to hide latency



## Tracking Branch Divergence

- Similar to vector processors but masks handled internally
  - No explicit mask register
  - Per warp stack that stores PCs and masks for "not taken" paths
- On a conditional branch
  - Push the current mask onto stack
  - Push the mask and PC for the "not taken" path
  - Set the mask for the "taken" path and execute
- At the end of the "taken" path
  - Pop the mask and PC for the "not taken" path and execute
- At the end of the "not taken" path
  - Pop the original mask before the branch instruction
- If a mask is all zeros, the block is skipped



## Tracking Branch Divergence

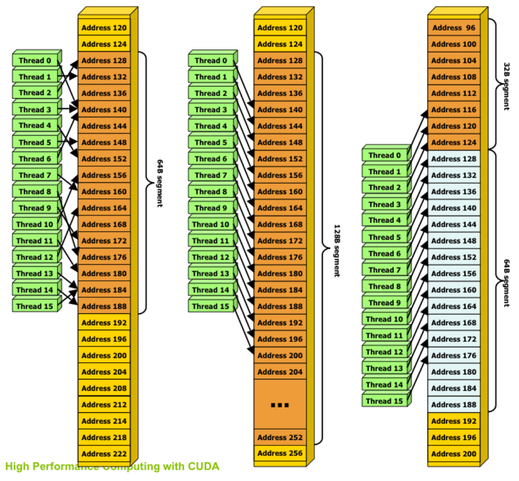
```
Mask = 1111              ──→  A = Data[i];
                              B = Data[j];

Push Mask = 1100         ──→  If (condition A)
Set Mask = 0011               {

Push Mask = 0010         ──→     if (condition B)
Set Mask = 0001                  {
                                    Y = A+B;
Pop Mask = 0010          ──→     } else {
                                    Y = A*B;
                                 }

Pop Mask = 1100          ──→  } else {
                                 Y = A-B;
Mask = 1111              ──→  }
```

## Memory System

- Per SM caches/memories
  - Instruction and constant caches
  - Multi-banked shared memory
- Distributed texture cache
  - Per TPC L1 and distributed L2 cache
  - Specialized for texture accesses in graphics pipeline
  - Moving towards generalized and shared L2 in upcoming chips
- Multi-channel DRAM main memory (e.g. 8 DDR-3 channels)
  - Interleaved addresses to achieve higher bandwidth
  - Lossless and lossy compression used to increase bandwidth
  - Aggressive access scheduling used to increase bandwidth
- Per thread private memory and global memory mapped to DRAM
  - Relying on threads to hide long latencies

## Memory Access Coalescing

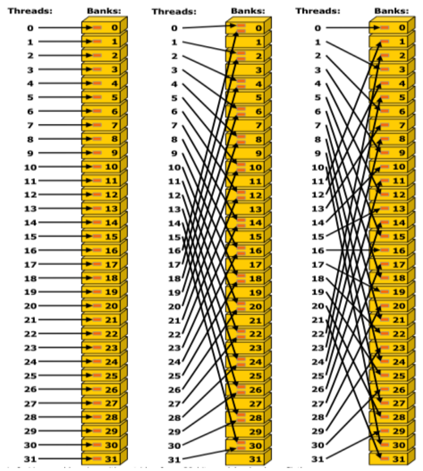
Goal: Combine multiple memory accesses generated from multiple threads into a single physical transaction



Paulius Micikeviciu, “M02: High Performance CompuIng with CUDA: OpImizing CUDA”, SC08, AusIn, TX.

## Memory Bank Conflicts

A bank conflict occurs if two or more threads access any bytes within different 32-bit words belonging to the Same bank.



NVIDIA CUDA C Programming Guide Version 4.0, Figure F-2

## Synchronization

- Barrier synchronization within a thread block
  - Tracking simplified by grouping threads into warps
  - Counter used to track number of threads that have arrived to barrier
- Atomic operations to global memory
  - Atomic read-modify-write (add, min, max, and, or, xor)
  - Atomic exchange or compare and swap
  - They are tied to DRAM latency
    - Moving to shared L2 in upcoming chips

## Intel Graphics ISA Syntax

- **[Predicate] Instr (ExecSize) Dst Src0 Src1 {Statement}**

- Predicate: +, -, Channel Replication (+x)
- Instr: opcode[.modifier][.flagmod]
  - Instruction modifier: add.sat
  - Flag modifier: mul.f0.g; add.f1.e
- Dst: RegFile RegNum<Region>.dmask:type
- Src0: (SrcMod) RegFile RegNum<Region>.swizzle:type
  - Source Modifier (SrcMod): -, (abs), -(abs)
- Src1: (SrcMod) RegFile RegNum<Region>.swizzle:type
- Example
  - **(+f0.0) add.f0.g (8) r2<8;8,1>:f -r10<8;8,1>:f -(abs) r12<8;8,1>:f**
  - Details later…

## Intel Graphics ISA

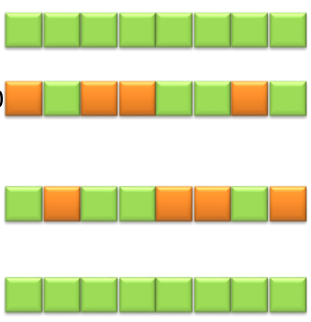
- Arithmetic
  - add, mul, mad, mac, …
- Bitwise
  - and, or, not, xor, shl, shr, bfe, …
- Control flow
  - if, else, endif, while, call, break, brc, jmpi, . . .
- Math
  - Inv, sin, cos, idiv, log, …
- Atomic
  - And, or, add, sub, inc, cmpwr,
- Barrier
- Load/store
- Sample
- Misc

## SIMD Control Flow

- HW support for SIMD control flow
- Execution paths of multiple threads start diverging
- Efficiency of throughput drops

Active channels 🟩 Inactive channels 🟧

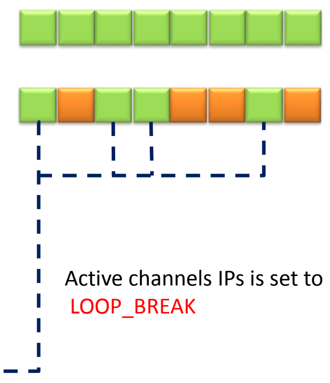
```
cmp.ne.f0.0 (8) null<1>:w r26.0<16;16,1>:w 0
(+f0) If  (8)  off_endif  off_else
   . . .
else (8)  off_endif
   . . .
endif  (8)  off_redirect
```

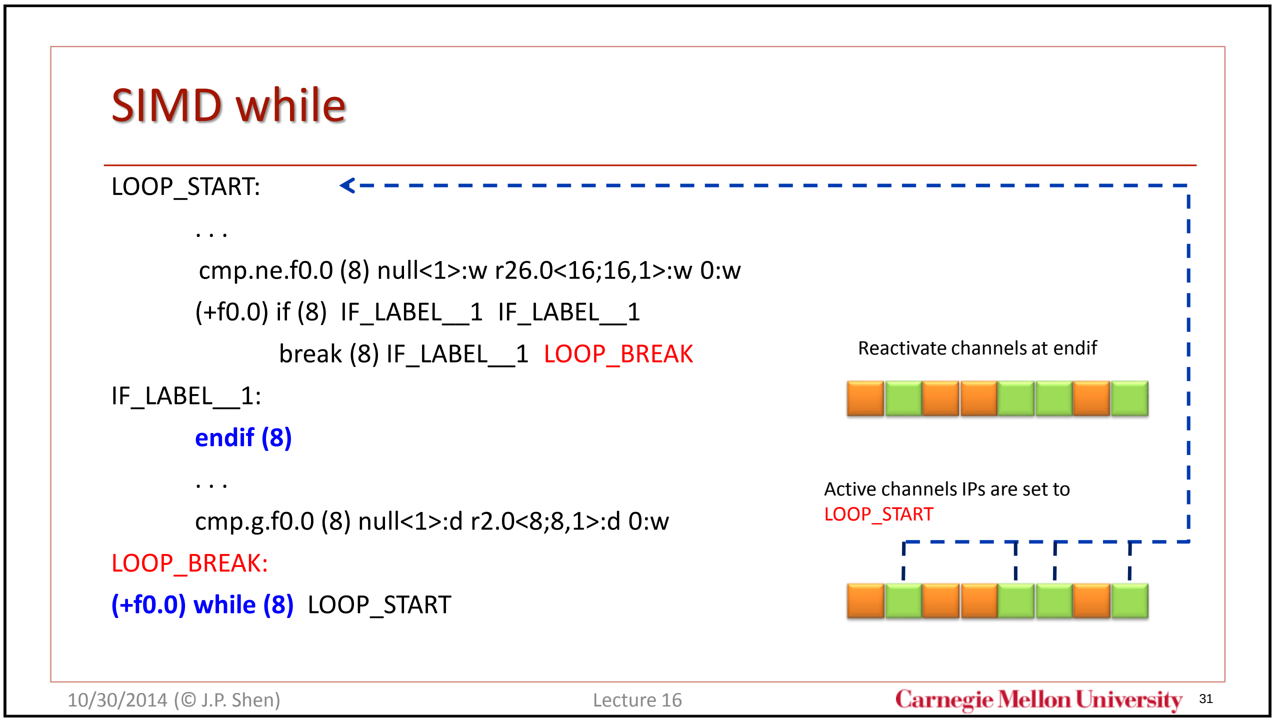


## SIMD while

Active channels 🟩 Inactive channels 🟧

```
LOOP_START:
          . . .
          cmp.ne.f0.0 (8) null<1>:w r26.0<16;16,1>:w 0:w
          (+f0.0) if (8)  IF_LABEL__1  IF_LABEL__1
                   break (8) IF_LABEL__1  LOOP_BREAK
IF_LABEL__1:
          endif (8)
          . . .
          cmp.g.f0.0 (8) null<1>:d r2.0<8;8,1>:d 0:w
LOOP_BREAK:
(+f0.0) while (8)  LOOP_START
```



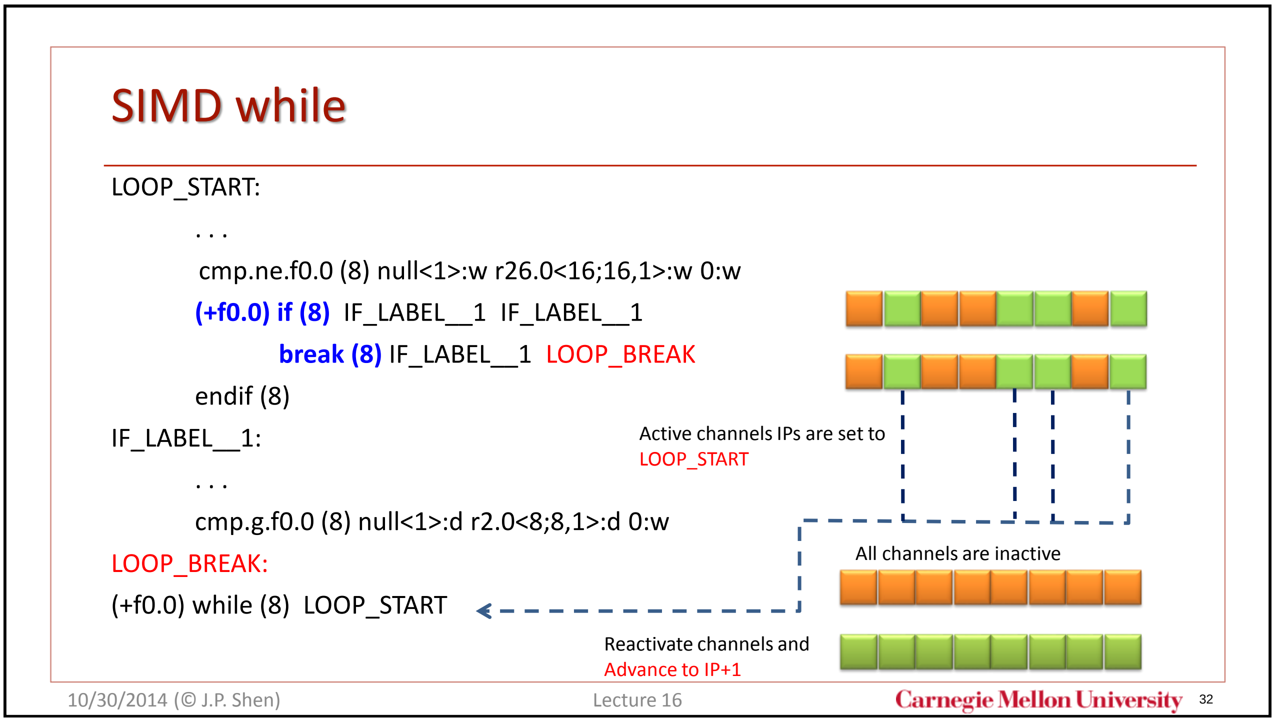


## SIMD while

```
LOOP_START:
        . . .
        cmp.ne.f0.0 (8) null<1>:w r26.0<16;16,1>:w 0:w
        (+f0.0) if (8)  IF_LABEL__1  IF_LABEL__1
                break (8) IF_LABEL__1  LOOP_BREAK
IF_LABEL__1:
        endif (8)
        . . .
        cmp.g.f0.0 (8) null<1>:d r2.0<8;8,1>:d 0:w
LOOP_BREAK:
(+f0.0) while (8)  LOOP_START
```

Reactivate channels at endif

Active channels IPs are set to LOOP_START



## SIMD while

```
LOOP_START:
        . . .
        cmp.ne.f0.0 (8) null<1>:w r26.0<16;16,1>:w 0:w
        (+f0.0) if (8)  IF_LABEL__1  IF_LABEL__1
                break (8) IF_LABEL__1  LOOP_BREAK
        endif (8)
IF_LABEL__1:
        . . .
        cmp.g.f0.0 (8) null<1>:d r2.0<8;8,1>:d 0:w
LOOP_BREAK:
(+f0.0) while (8)  LOOP_START
```

Active channels IPs are set to LOOP_START

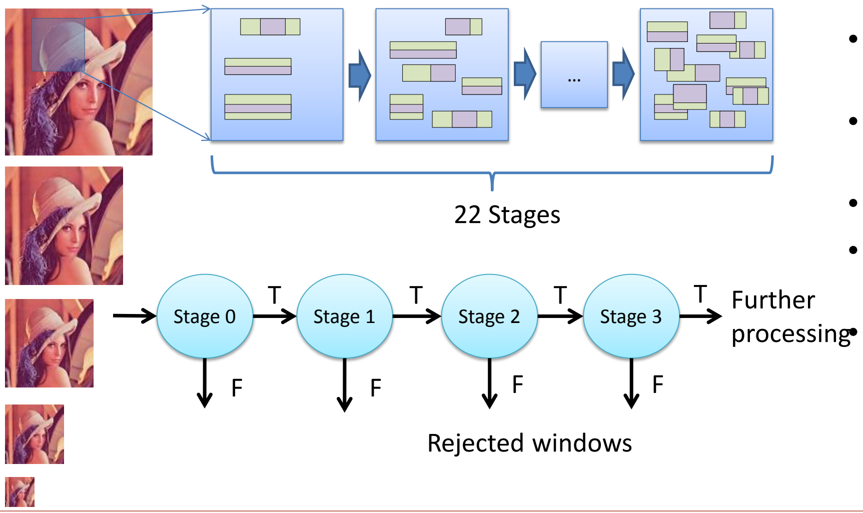
All channels are inactive

Reactivate channels and Advance to IP+1

## Fundamental Challenges of GPGPU

- Diverging control flow
- Irregular memory access
- Load balancing between CPU and GPU

## Case Study: Viola-Jones Face Detector



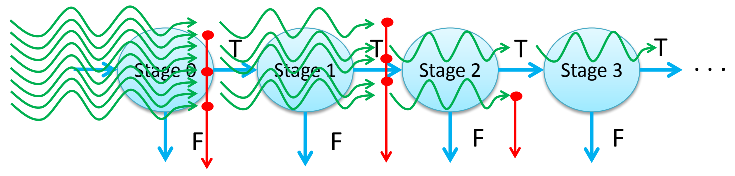
- Based on Harr-like feature
- 24x24 detecting window
- Scaling is 1.2
- Check feature against threshold
- Step 2 pixels
  - 512x512 image needs ~60K sub windows

## Diverging Control Flow

**Pseudo Code:**

```
for i in n_stages:
    for f in stage_features[i]:
        sum += weakClassifier(f)

    If sum < stage_threshold[i]:
        return False
return True
```

SIMD8: 8 sub windows

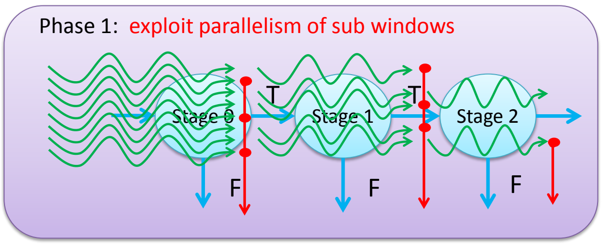


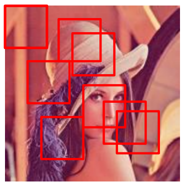
**Efficiency drops to 12.5%**

| Stage | 1 | 2 | 3 | 4 | 5 | 6 | ... | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|---|
| HAAR Feature Count | 3 | 16 | 21 | 39 | 33 | 44 | | 182 | 211 | 213 |
| Eliminate Averagely (%) | 20.41 | 30.04 | 25.84 | 11.95 | 3.32 | 3.07 | | 0.02 | 0.01 | 0.01 |

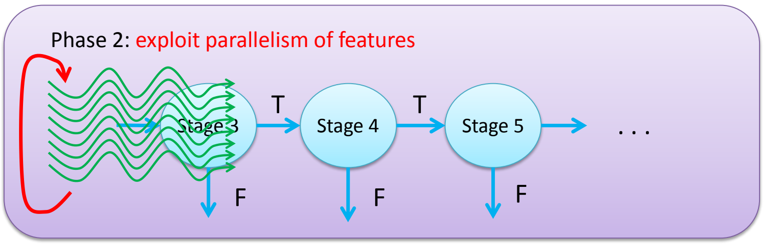
First **3** stages eliminate 75+% sub windows on average

## Divide Into Phases & Exploit Different Parallelism

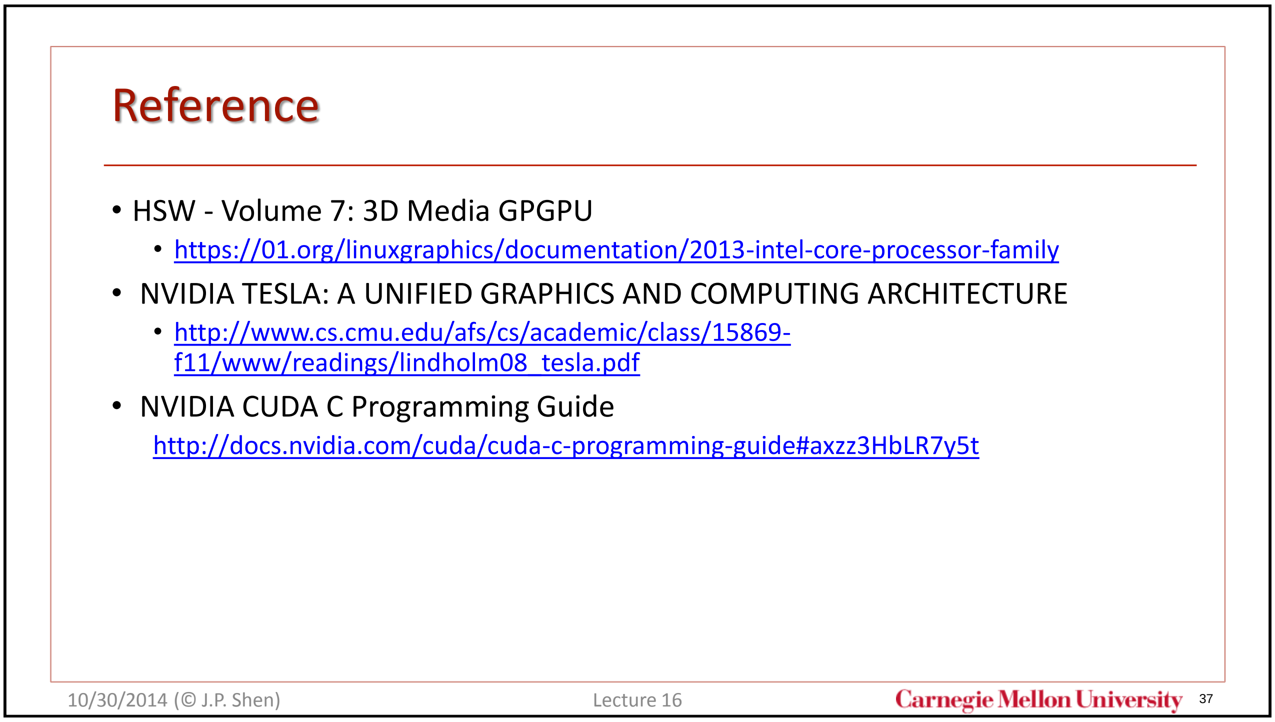




Gather positive sub windows



- Each stage has enough features
- Iterate through 8 features at one time
- High efficiency



## Reference

- HSW - Volume 7: 3D Media GPGPU
  - https://01.org/linuxgraphics/documentation/2013-intel-core-processor-family
- NVIDIA TESLA: A UNIFIED GRAPHICS AND COMPUTING ARCHITECTURE
  - http://www.cs.cmu.edu/afs/cs/academic/class/15869-f11/www/readings/lindholm08_tesla.pdf
- NVIDIA CUDA C Programming Guide

  http://docs.nvidia.com/cuda/cuda-c-programming-guide#axzz3HbLR7y5t