CS 152 Computer Architecture and Engineering

Lecture 16: Graphics Processing Units (GPUs)

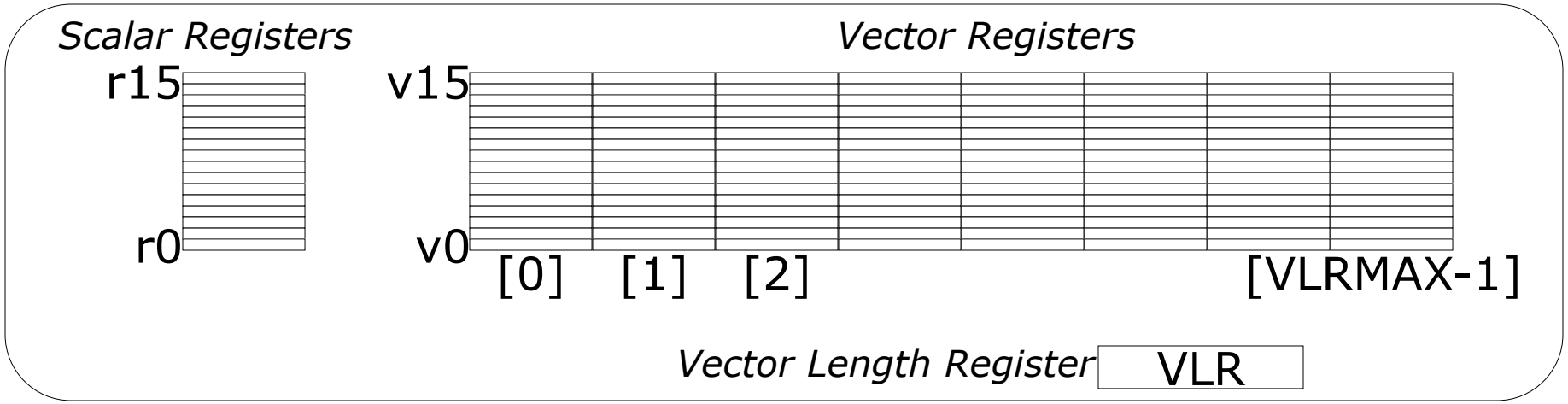
Dr. George Michelogiannakis
EECS, University of California at Berkeley
CRD, Lawrence Berkeley National Laboratory

http://inst.eecs.berkeley.edu/~cs152

Administrivia

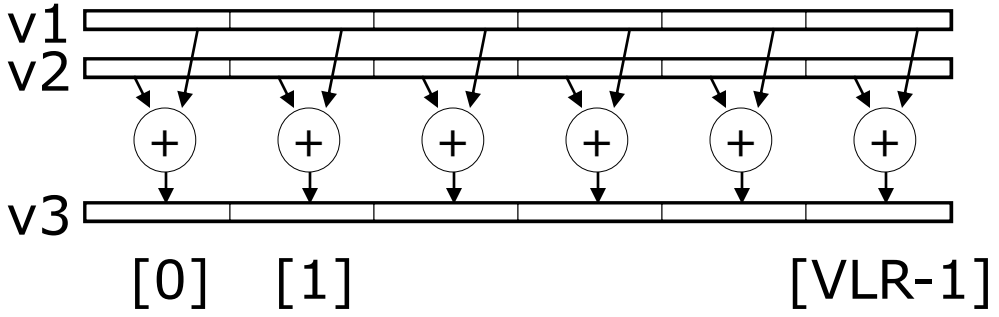
- PS5 is out
- PS4 due on Wednesday
- Lab 4
- Quiz 4 on Monday April 11th

Vector Programming Model



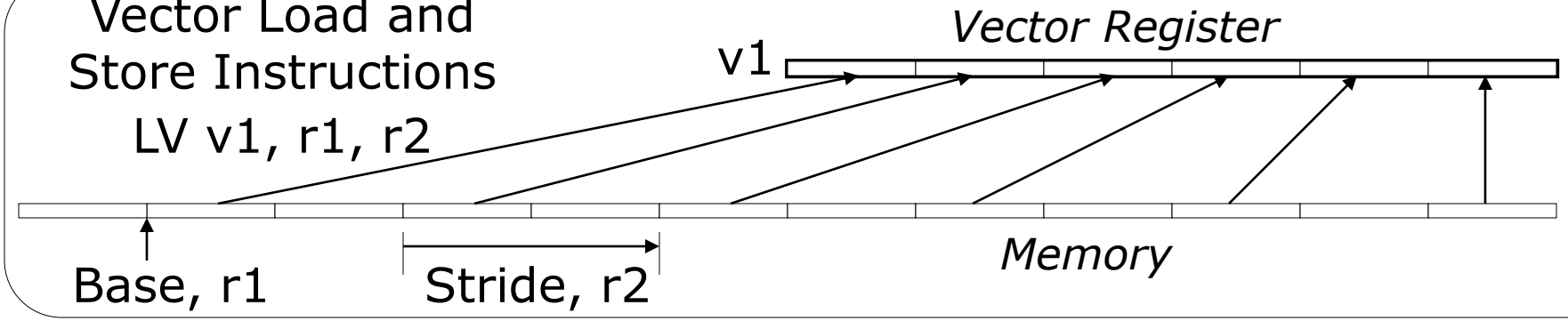
Vector Arithmetic Instructions

ADDV v3, v1, v2



Vector Load and Store Instructions

LV v1, r1, r2

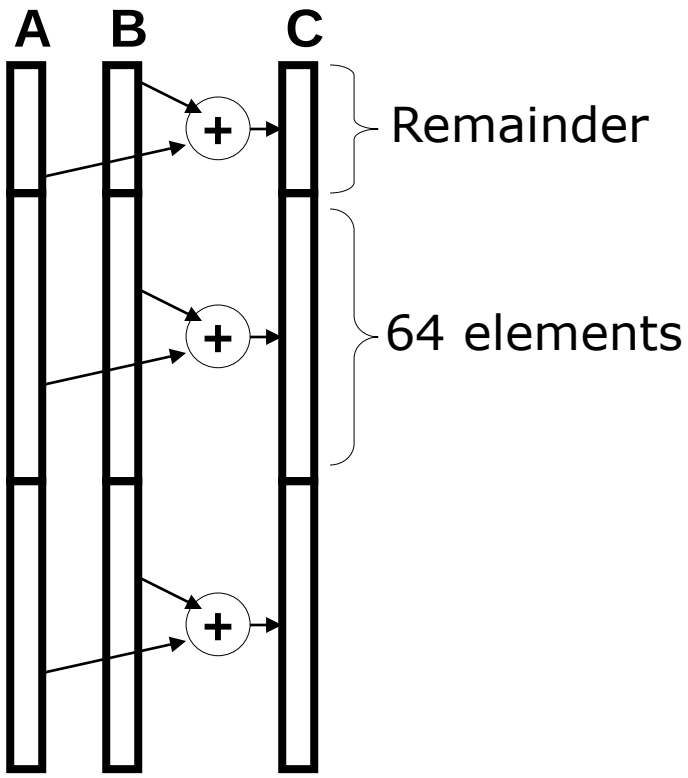


Vector Stripmining

Problem: Vector registers have finite length

Solution: Break loops into pieces that fit in registers, *"Stripmining"*

```
for (i=0; i<N; i++)
    C[i] = A[i]+B[i];
```



Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:

```
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

Solution: Add vector *mask* (or *flag*) registers

- vector version of predicate registers, 1 bit per element

…and *maskable* vector instructions

- vector operation becomes bubble ("NOP") at elements where mask bit is clear

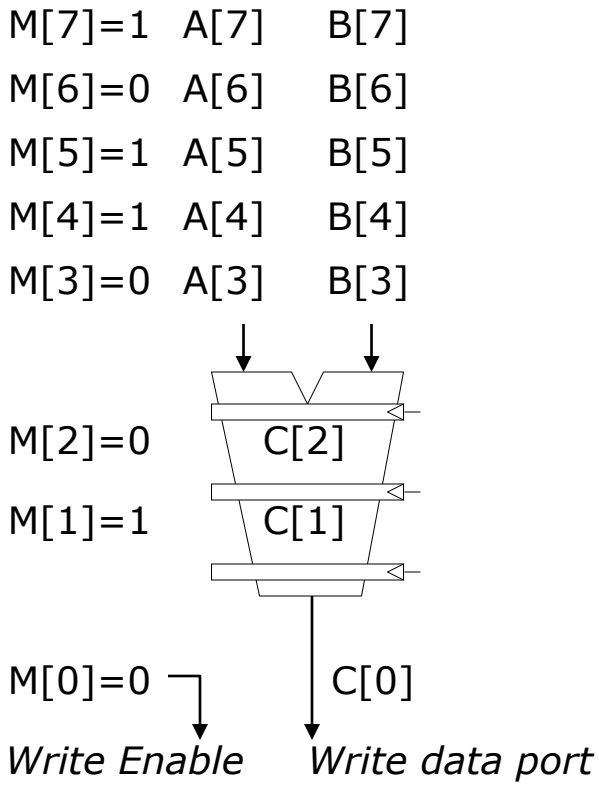
Code example:

```
CVM              # Turn on all elements
LV vA, rA        # Load entire A vector
SGTVS.D vA, F0   # Set bits in mask register where A>0
LV vA, rB        # Load B vector into A under mask
SV vA, rA        # Store A back to memory under mask
```

Masked Vector Instructions

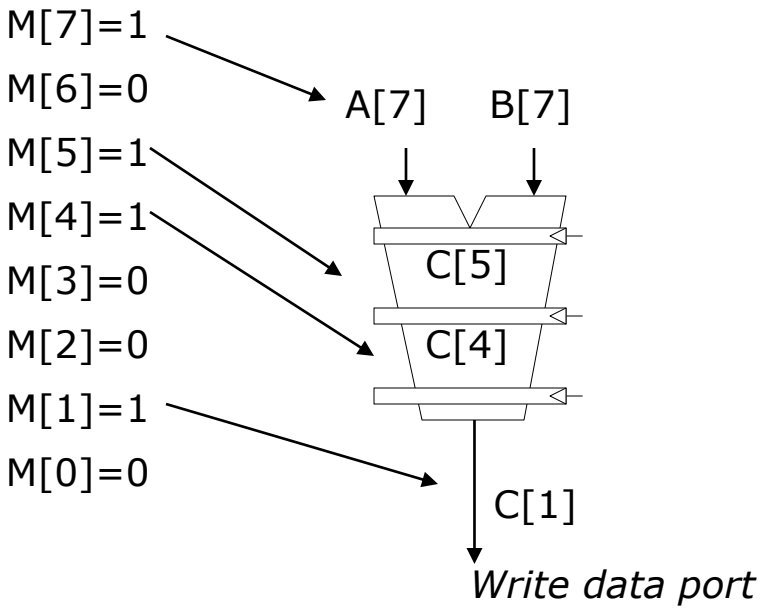
Simple Implementation

- execute all N operations, turn off result writeback according to mask



Density-Time Implementation

- scan mask vector and only execute elements with non-zero masks



Vector Reductions

Problem: Loop-carried dependence on reduction variables

```
sum = 0;
for (i=0; i<N; i++)
    sum += A[i];  # Loop-carried dependence on sum
```

Solution: Re-associate operations if possible, use binary tree to perform reduction

```
# Rearrange as:
sum[0:VL-1] = 0                     # Vector of VL partial sums
for(i=0; i<N; i+=VL)                # Stripmine VL-sized chunks
    sum[0:VL-1] += A[i:i+VL-1]; # Vector sum
# Now have VL partial sums in one vector register
do {
    VL = VL/2;                        # Halve vector length
    sum[0:VL-1] += sum[VL:2*VL-1] # Halve no. of partials
} while (VL>1)
```

Vector Scatter/Gather

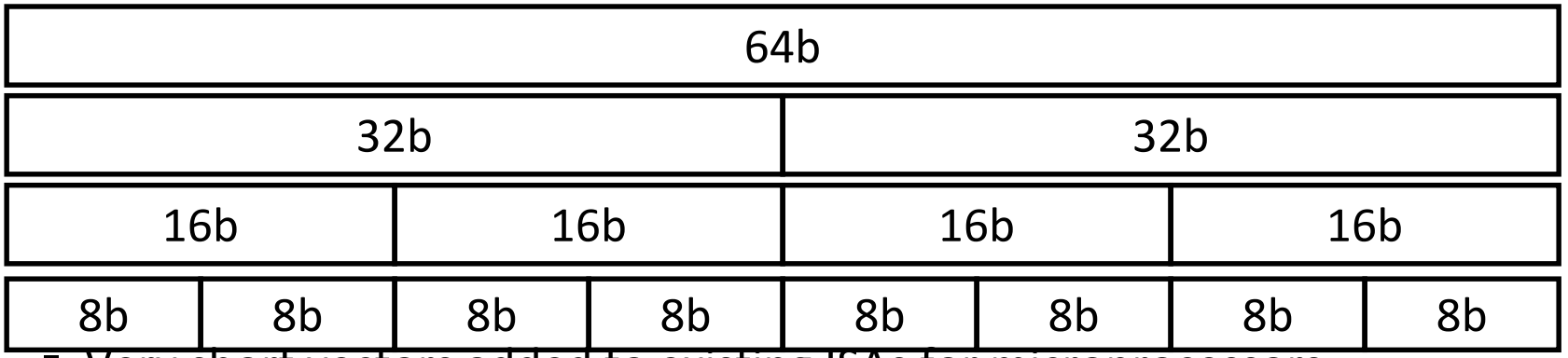
Want to vectorize loops with indirect accesses:

```
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

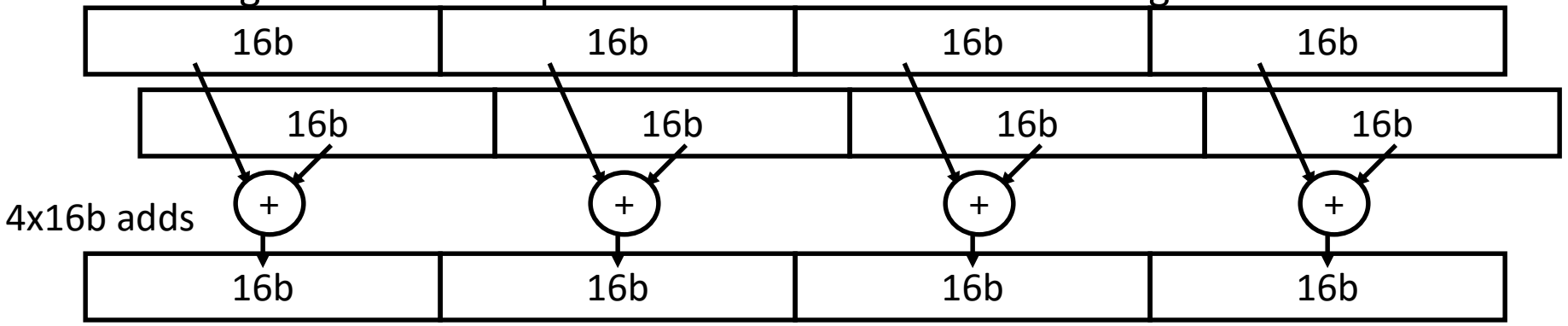
Indexed load instruction (*Gather*)

```
LV vD, rD         # Load indices in D vector
LVI vC, rC, vD    # Load indirect from rC base
LV vB, rB         # Load B vector
ADDV.D vA,vB,vC   # Do add
SV vA, rA         # Store result
```

Multimedia Extensions (aka SIMD extensions)



- Very short vectors added to existing ISAs for microprocessors
- Use existing 64-bit registers split into 2x32b or 4x16b or 8x8b
  - Lincoln Labs TX-2 from 1957 had 36b datapath split into 2x18b or 4x9b
  - Newer designs have wider registers
    - 128b for PowerPC Altivec, Intel SSE2/3/4
    - 256b for Intel AVX
- Single instruction operates on all elements within register



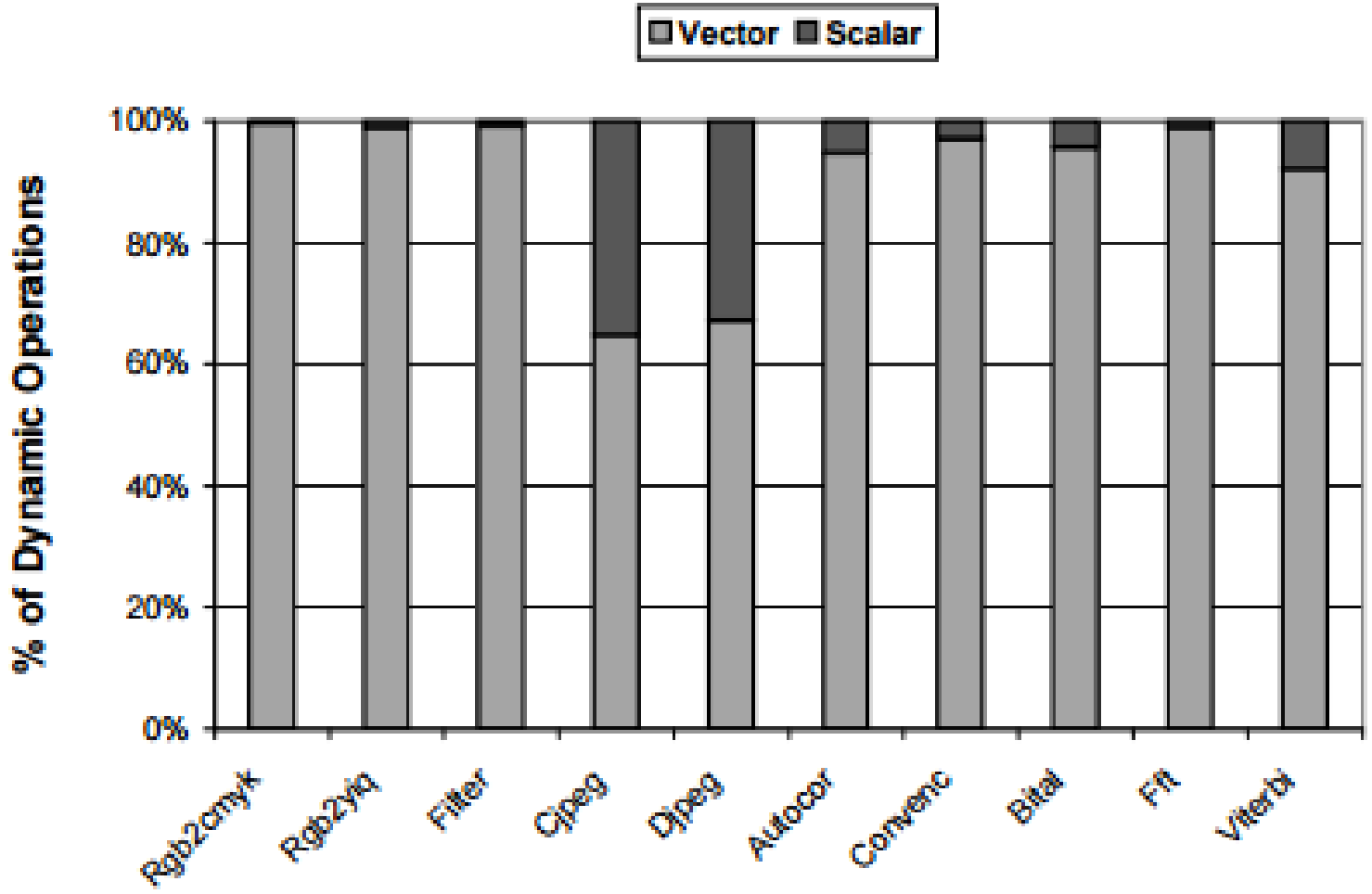
Multimedia Extensions versus Vectors

- Limited instruction set:
  - no vector length control
  - no strided load/store or scatter/gather
  - unit-stride loads must be aligned to 64/128-bit boundary
- Limited vector register length:
  - requires superscalar dispatch to keep multiply/add/load units busy
  - loop unrolling to hide latencies increases register pressure
- Trend towards fuller vector support in microprocessors
  - Better support for misaligned memory accesses
  - Support of double-precision (64-bit floating-point)
  - New Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b)

Degree of Vectorization

MIPS processor with vector coprocessor

- Compilers are good at finding data-level parallelism



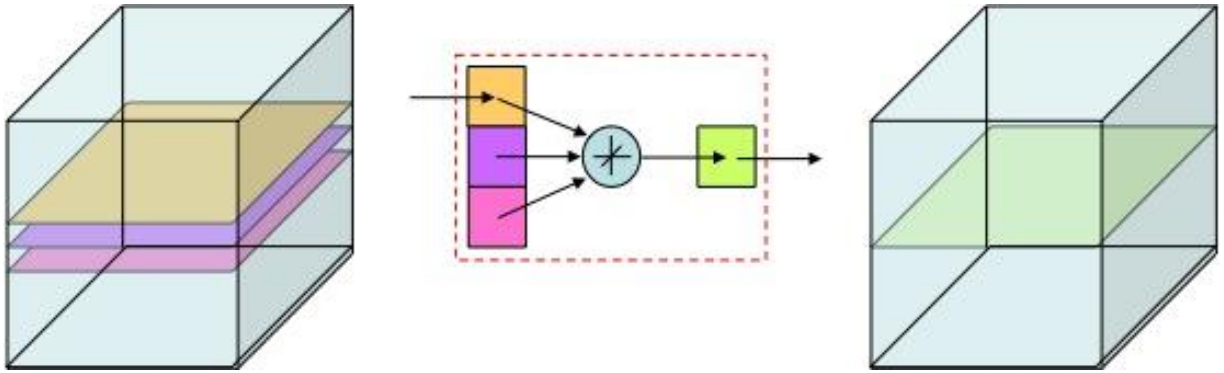
Types of Parallelism

- Instruction-Level Parallelism (ILP)
  - Execute independent instructions from one instruction stream in parallel (pipelining, superscalar, VLIW)
- Thread-Level Parallelism (TLP)
  - Execute independent instruction streams in parallel (multithreading, multiple cores)
- Data-Level Parallelism (DLP)
  - Execute multiple operations of the same type in parallel (vector/SIMD execution)

- Which is easiest to program?
- Which is most flexible form of parallelism?
  - i.e., can be used in more situations
- Which is most efficient?
  - i.e., greatest tasks/second/area, lowest energy/task

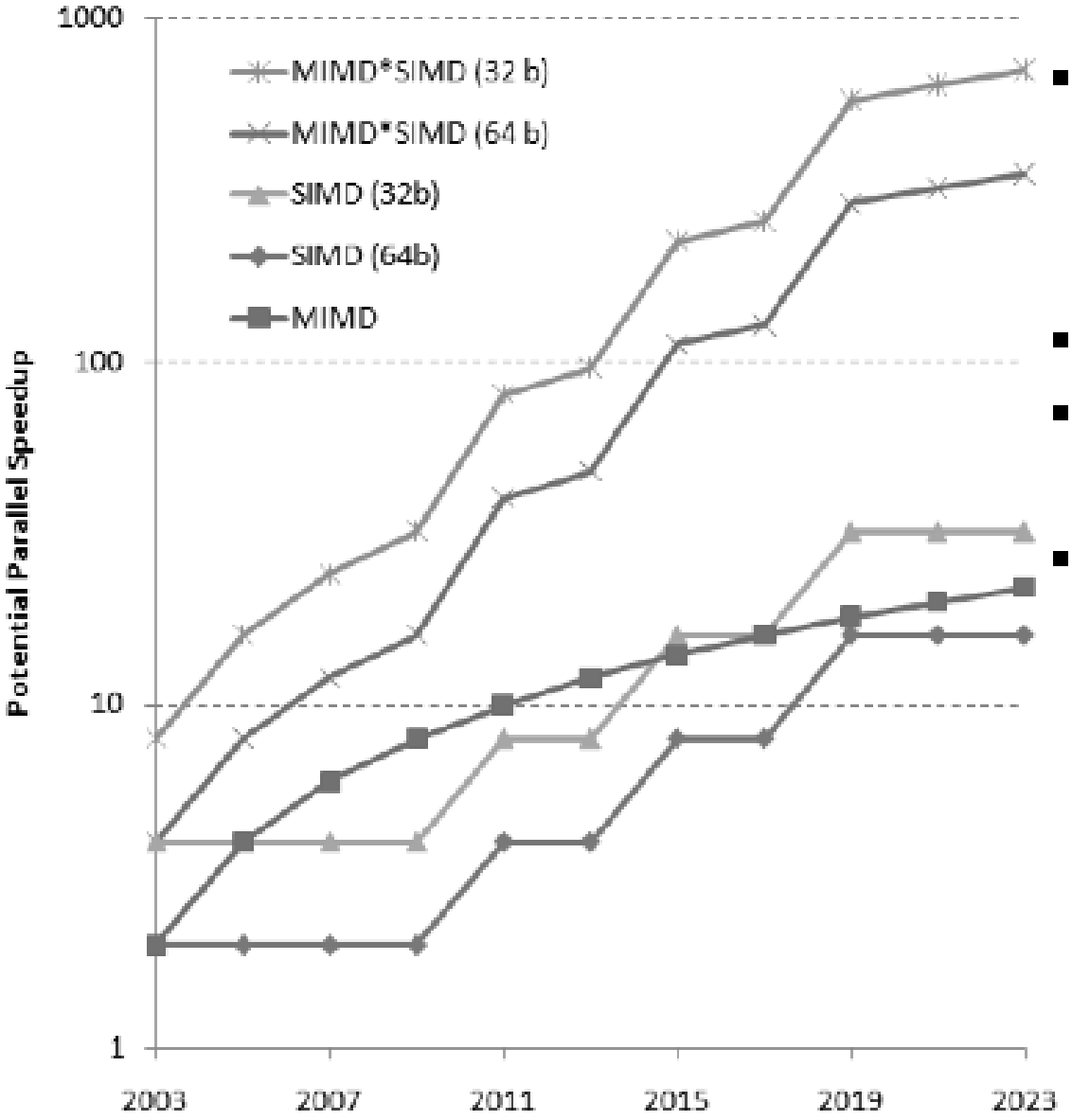
Resurgence of DLP

- Convergence of application demands and technology constraints drives architecture choice
- New applications, such as graphics, machine vision, speech recognition, machine learning, etc. all require large numerical computations that are often trivially data parallel



- SIMD-based architectures (vector-SIMD, subword-SIMD, SIMT/GPUs) are most efficient way to execute these algorithms

DLP important for conventional CPUs too



- Prediction for x86 processors, from Hennessy & Patterson, 5th edition
  - ***Note: Educated guess, not Intel product plans!***
- TLP: 2+ cores / 2 years
- DLP: 2x width / 4 years
- DLP will account for more mainstream parallelism growth than TLP in next decade.
  - SIMD –single-instruction multiple-data (DLP)
  - MIMD- multiple-instruction multiple-data (TLP)

Graphics Processing Units (GPUs)

- Original GPUs were dedicated fixed-function devices for generating 3D graphics (mid-late 1990s) including high-performance floating-point units
  - Provide workstation-like graphics for PCs
  - User could configure graphics pipeline, but not really program it
- Over time, more programmability added (2001-2005)
  - E.g., New language Cg for writing small programs run on each vertex or each pixel, also Windows DirectX variants
  - Massively parallel (millions of vertices or pixels per frame) but very constrained programming model
- Some users noticed they could do general-purpose computation by mapping input and output data to images, and computation to vertex and pixel shading computations
  - Incredibly difficult programming model as had to use graphics pipeline model for general computation

General-Purpose GPUs (GP-GPUs)

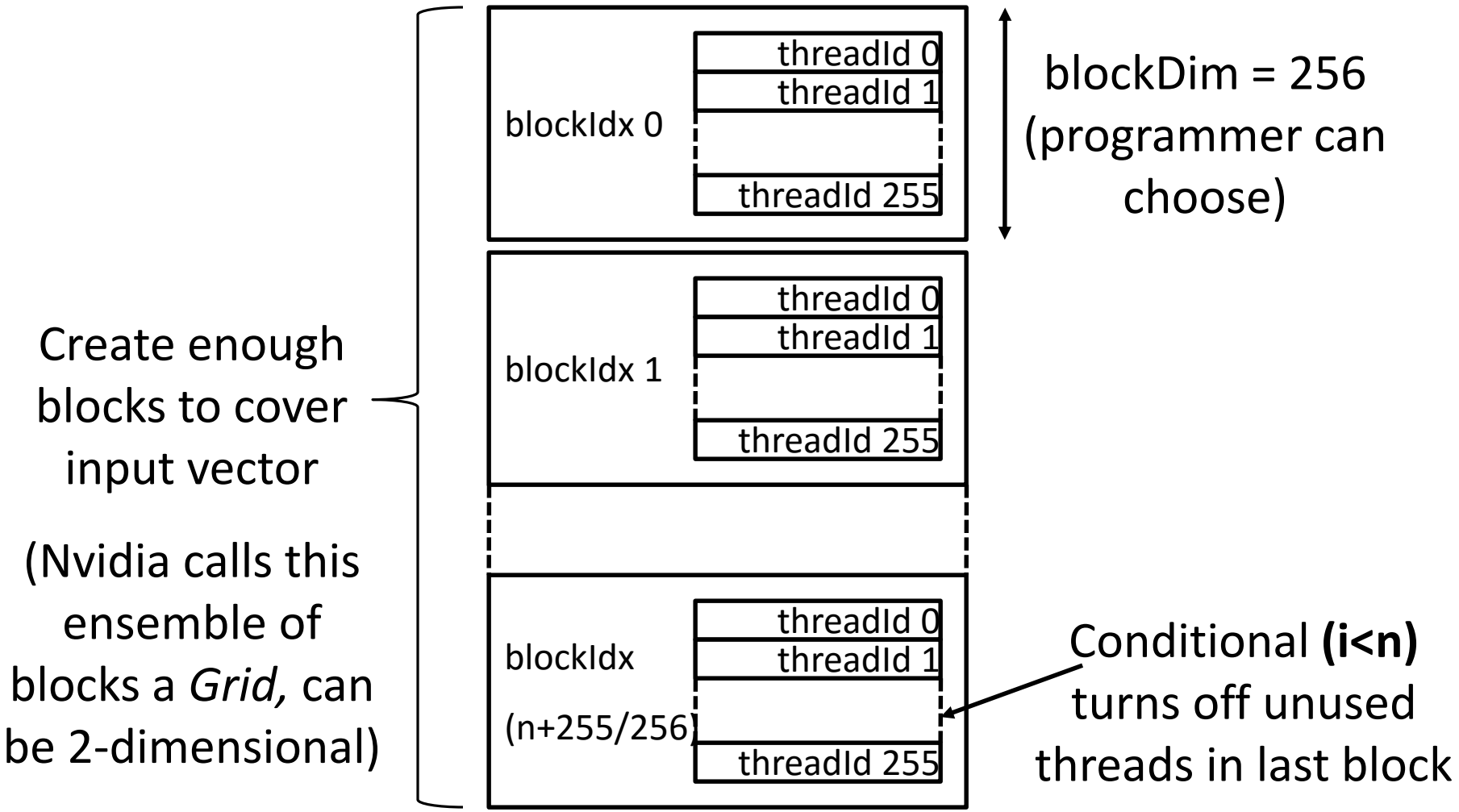
- In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language: CUDA
  - "Compute Unified Device Architecture"
  - Subsequently, broader industry pushing for OpenCL, a vendor-neutral version of same ideas.
- Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general-purpose computing
- Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution
- This lecture has a simplified version of Nvidia CUDA-style model and only considers GPU execution for computational kernels, not graphics
  - Would probably need another course to describe graphics processing

Simplified CUDA Programming Model

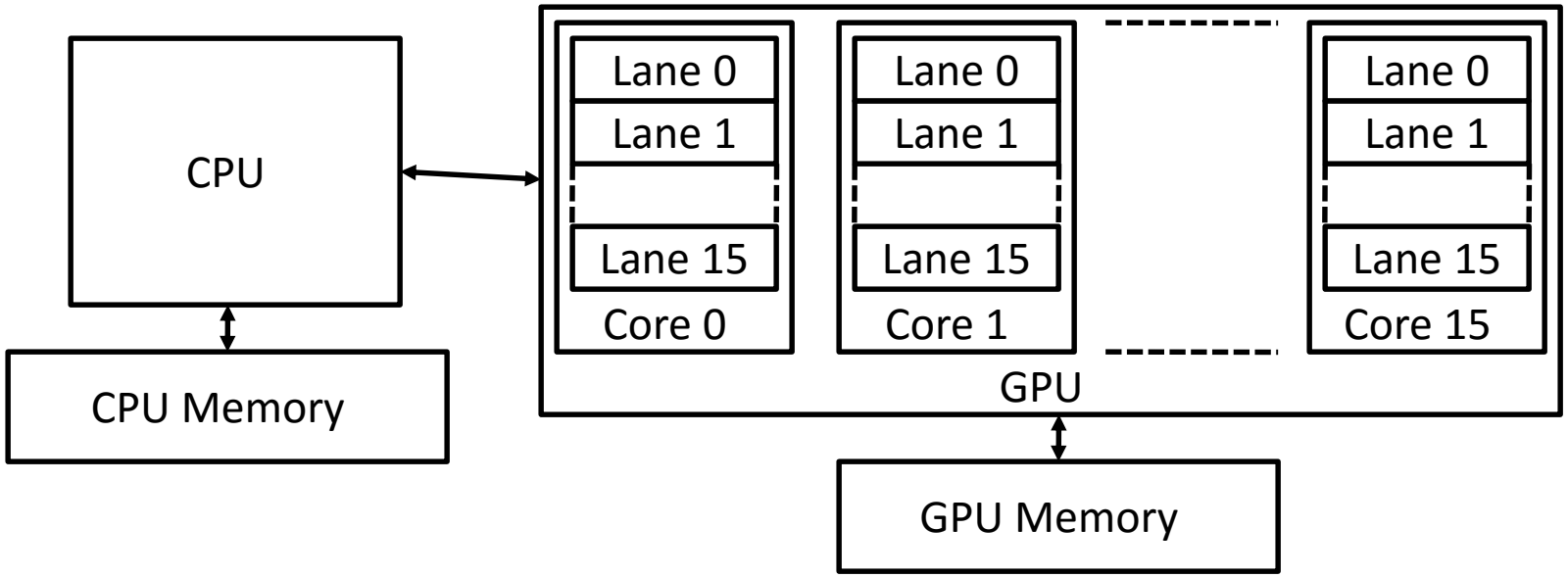
- Computation performed by a very large number of independent small scalar threads (*CUDA threads* or *microthreads*) grouped into *thread blocks.*

```
// C version of DAXPY loop.
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 version.
__host__   // Piece run on host processor.
int nblocks = (n+255)/256; // 256 CUDA threads/block
daxpy<<<nblocks,256>>>(n,2.0,x,y);

__device__   // Piece run on GP-GPU.
void daxpy(int n, double a, double*x, double*y)
{ int i = blockIdx.x*blockDim.x + threadId.x;
  if (i<n) y[i]=a*x[i]+y[i]; }
```

Programmer's View of Execution



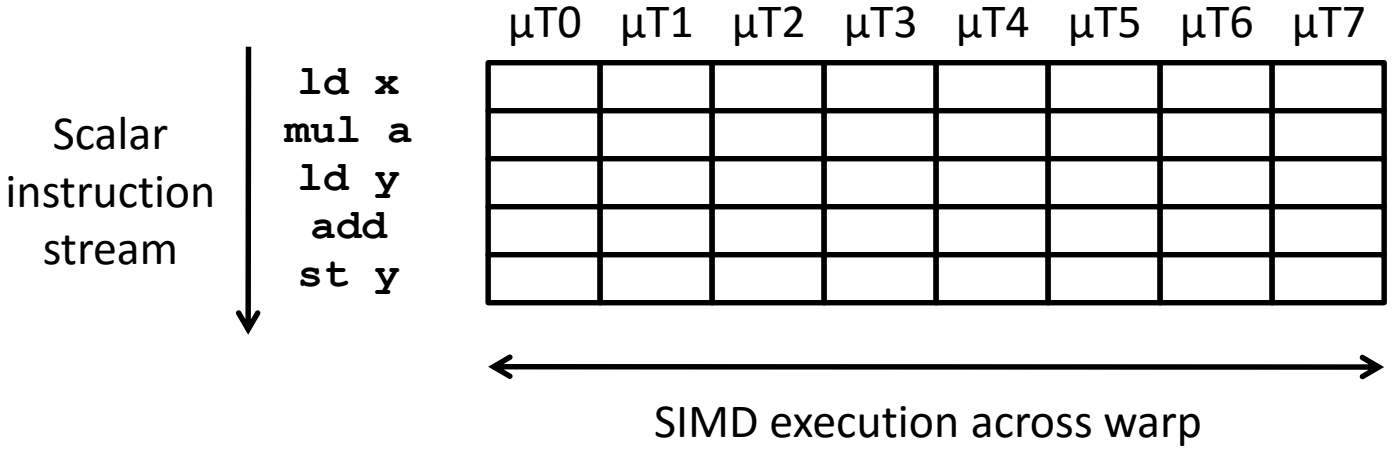
Hardware Execution Model



- GPU is built from multiple parallel cores, each core contains a multithreaded SIMD processor with multiple lanes but with no scalar processor
- CPU sends whole "grid" over to GPU, which distributes thread blocks among cores (each thread block executes on one core)
  - Programmer unaware of number of cores

"Single Instruction, Multiple Thread"

- GPUs use a SIMT model (SIMD with multithreading)
- Individual scalar instruction streams for each CUDA thread are grouped together for SIMD execution (each thread executes the same instruction each cycle) on hardware (Nvidia groups 32 CUDA threads into a *warp*). Threads are independent from each other



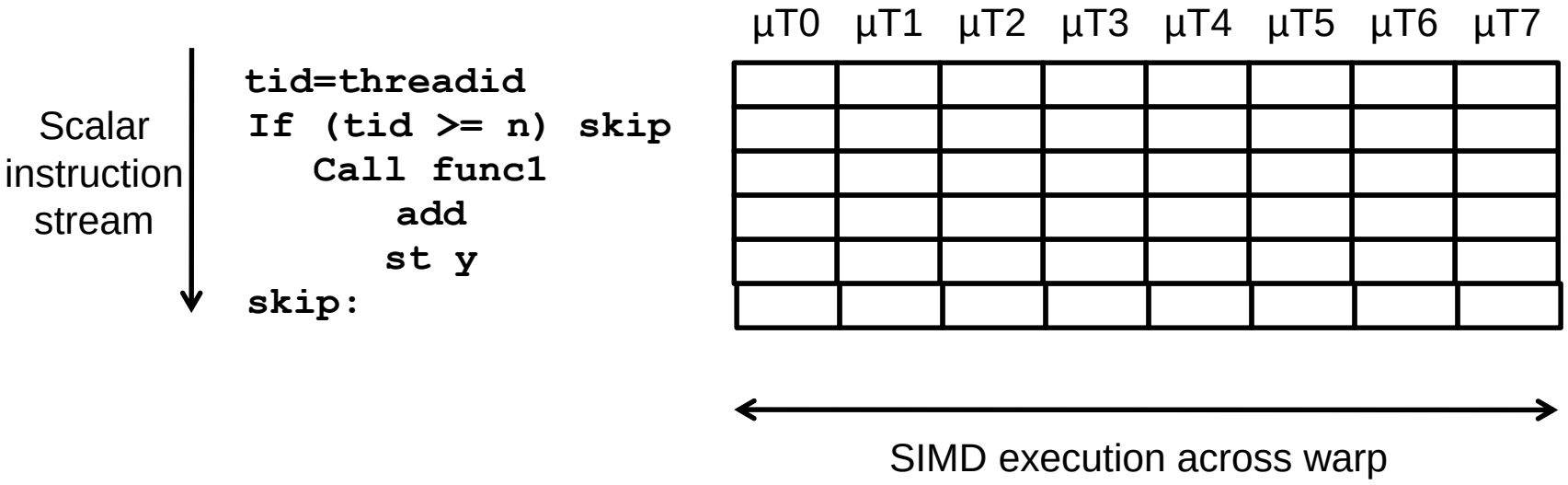
Implications of SIMT Model

- All “vector” loads and stores are scatter-gather, as individual µthreads perform scalar loads and stores
  - GPU adds hardware to dynamically coalesce individual µthread loads and stores to mimic vector loads and stores

- Every µthread has to perform stripmining calculations redundantly (“am I active?”) as there is no scalar processor equivalent

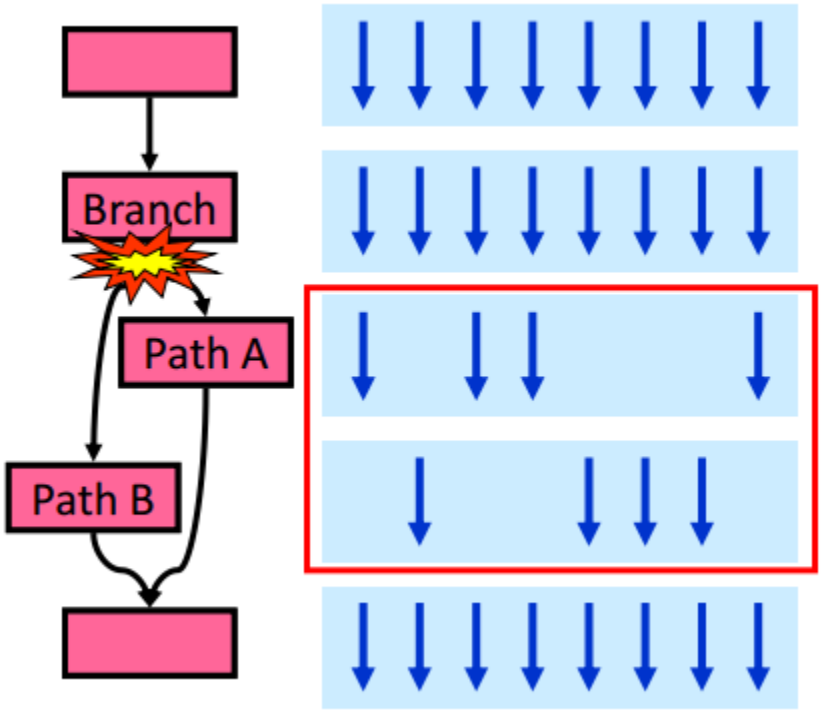
Conditionals in SIMT model

- Simple if-then-else are compiled into predicated execution, equivalent to vector masking
- More complex control flow compiled into branches
- How to execute a vector of branches?

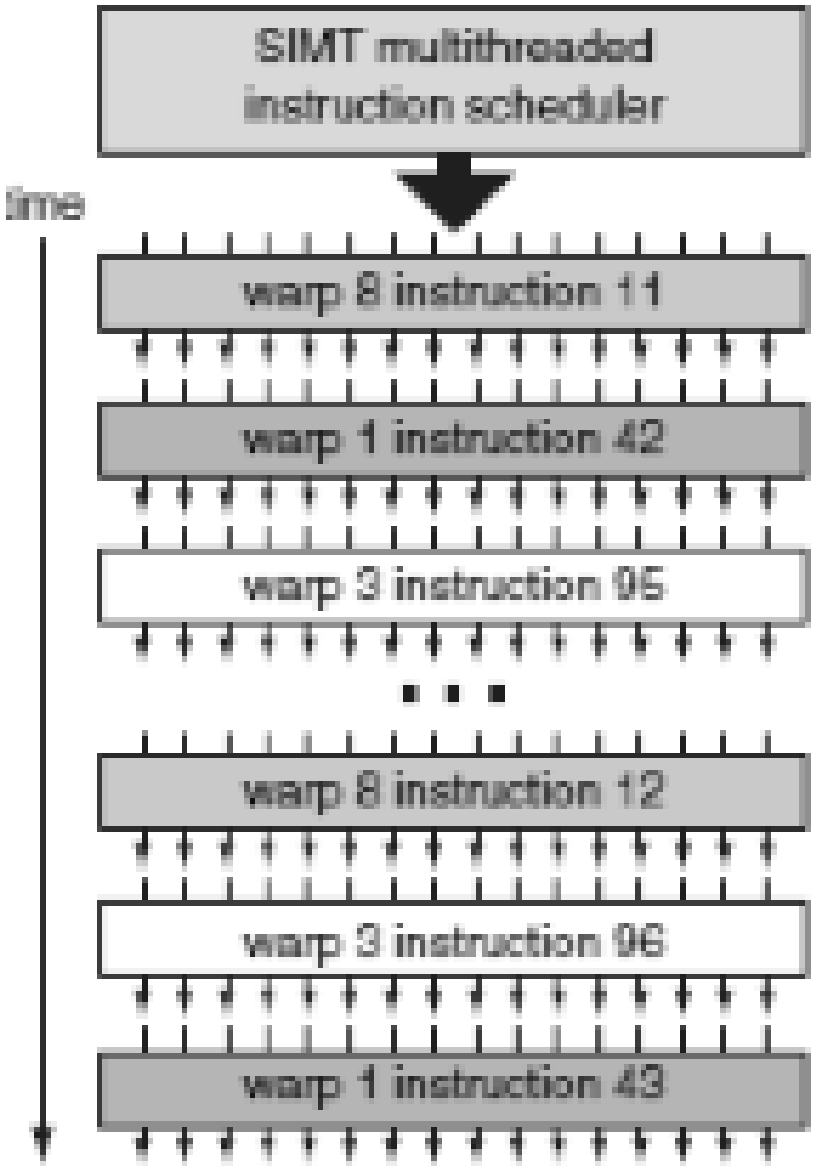


Branch divergence

- Hardware tracks which μthreads take or don't take branch
- If all go the same way, then keep going in SIMD fashion
- If not, create mask vector indicating taken/not-taken
- Keep executing not-taken path under mask, push taken branch PC+mask onto a hardware stack and execute later
- When can execution of μthreads in warp reconverge?



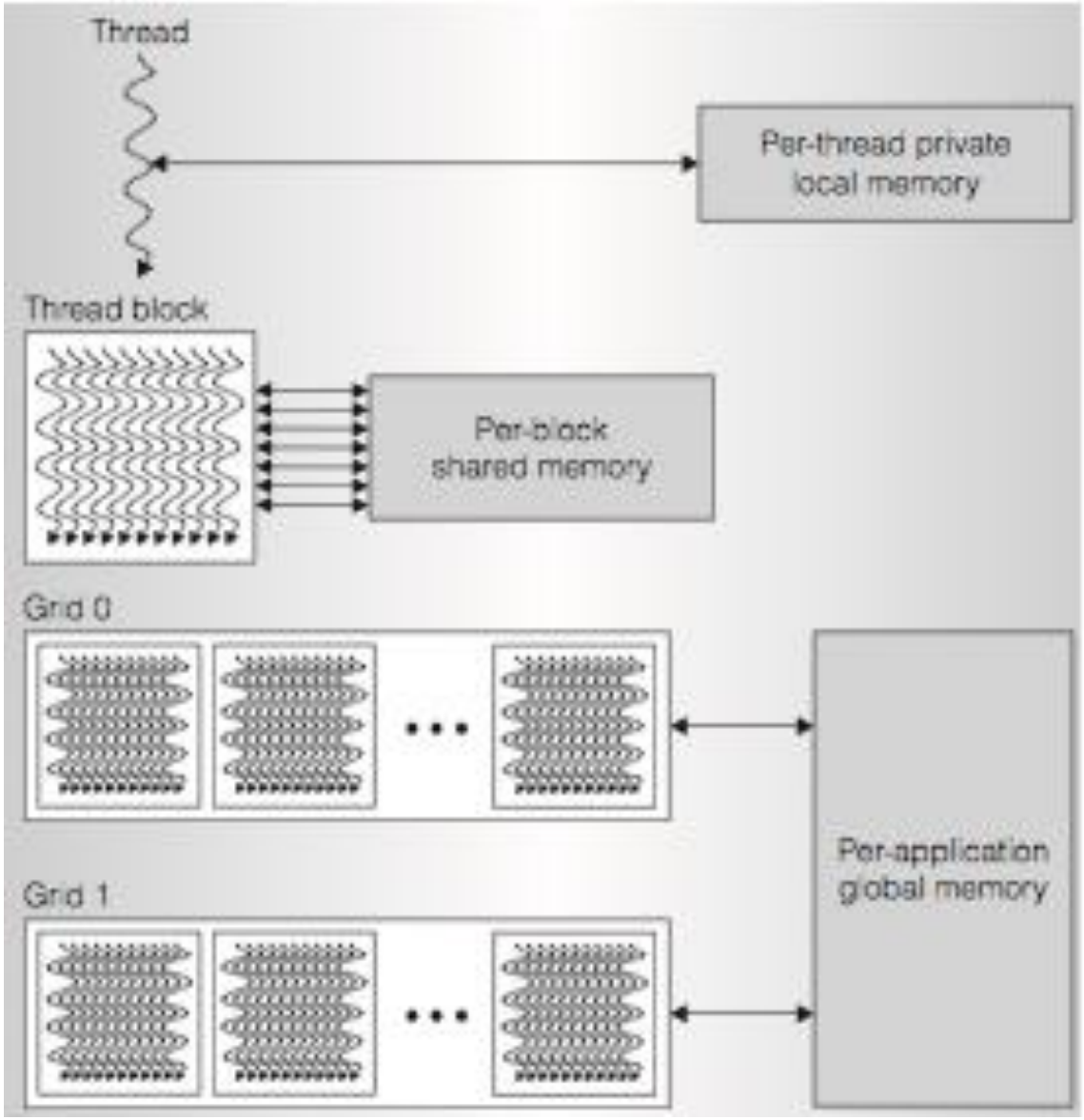
Warps are multithreaded on core



- One warp of 32 µthreads is a single thread in the hardware
- Multiple warp threads are interleaved in execution on a single core to hide latencies (memory and functional unit)
- A single thread block can contain multiple warps (up to 512 µT max in CUDA), all mapped to single core
- Can have multiple blocks executing on one core

[Nvidia, 2010]

GPU Memory Hierarchy



[Nvidia, 2010]

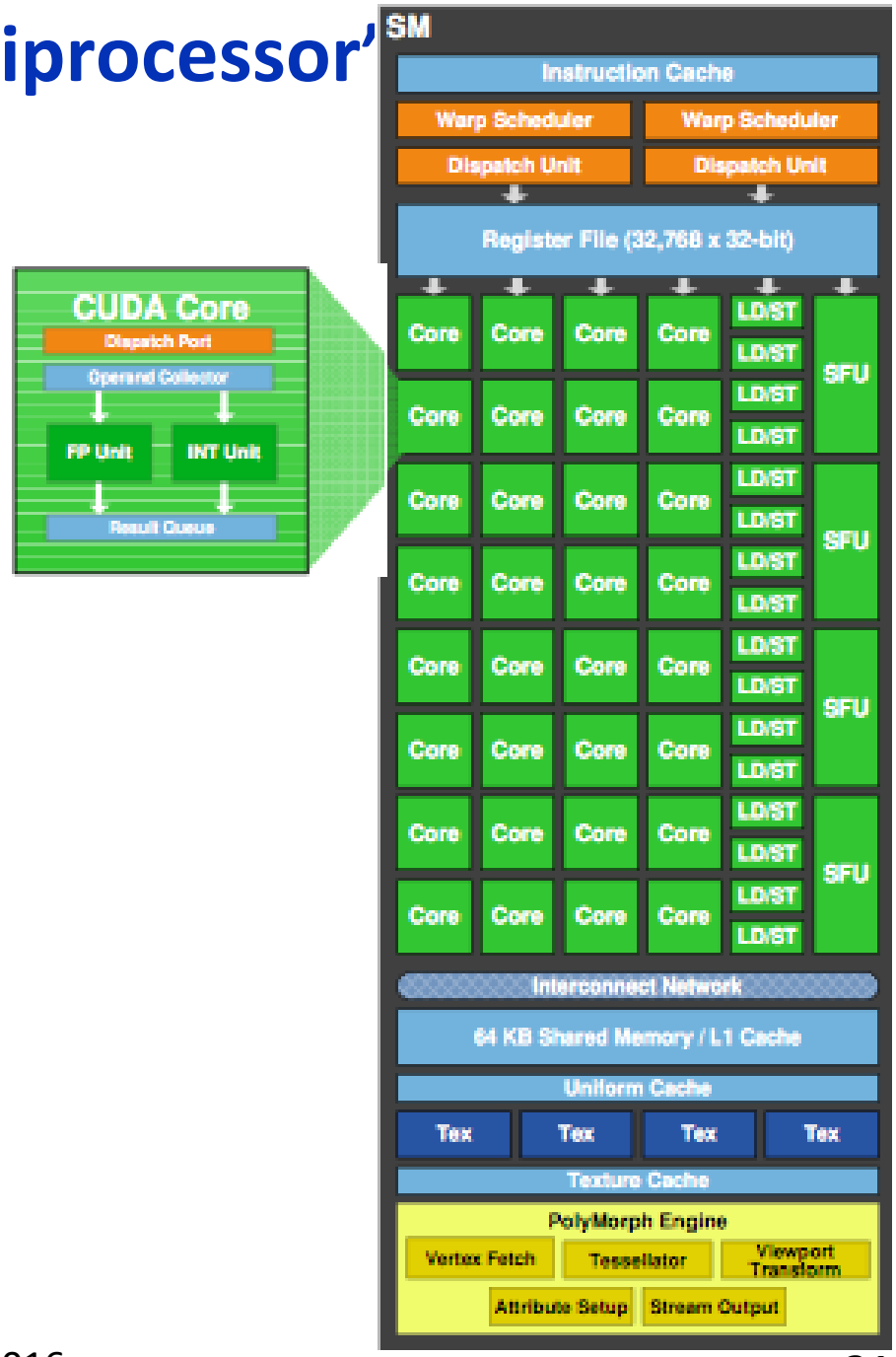
SIMT

- Illusion of many independent threads
  - Threads inside a warp execute in a SIMD fashion
- But for efficiency, programmer must try and keep µthreads aligned in a SIMD fashion
  - Try and do unit-stride loads and store so memory coalescing kicks in
  - Avoid branch divergence so most instruction slots execute useful work and are not masked off

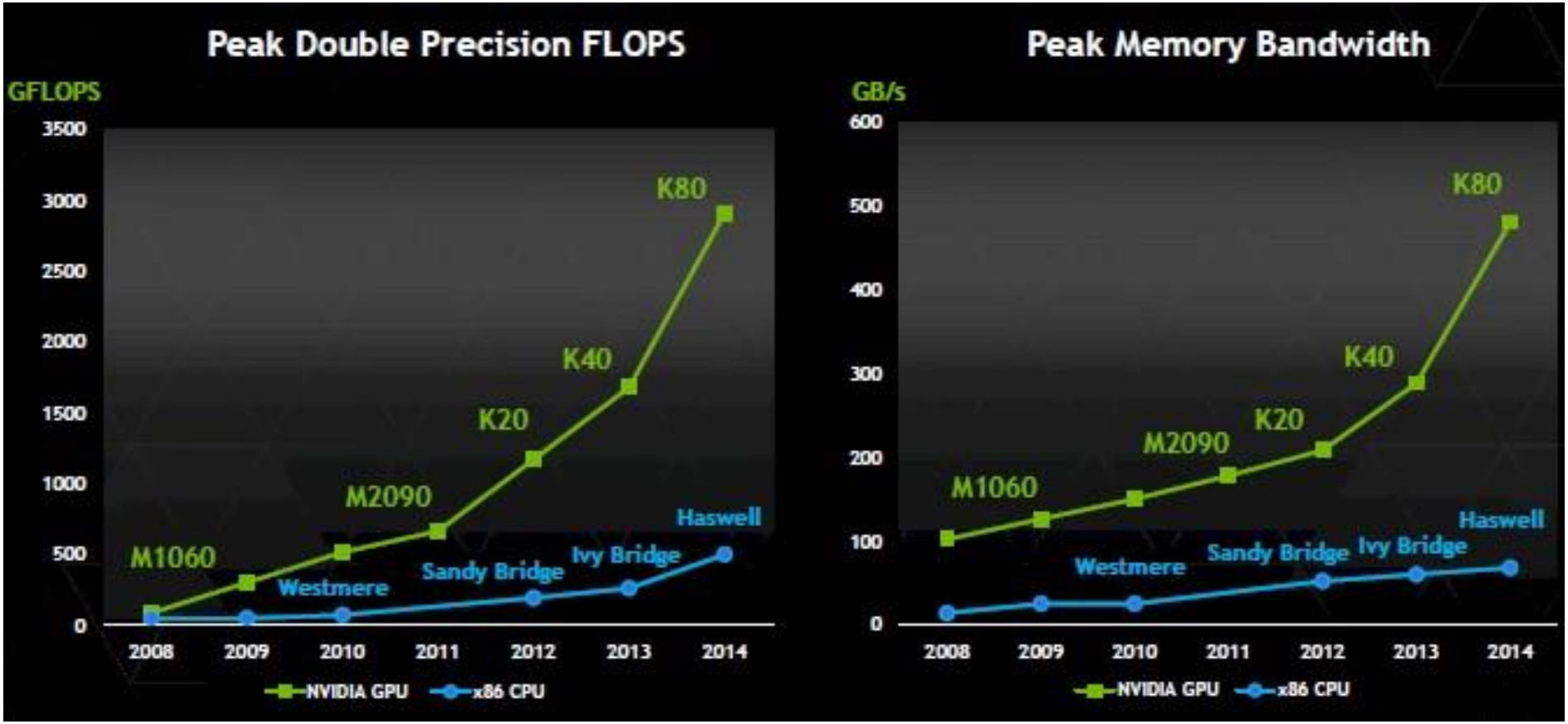
Nvidia Fermi GF100 GPU

[Nvidia, 2010]

Fermi "Streaming Multiprocessor" Core



GPU Versus CPU



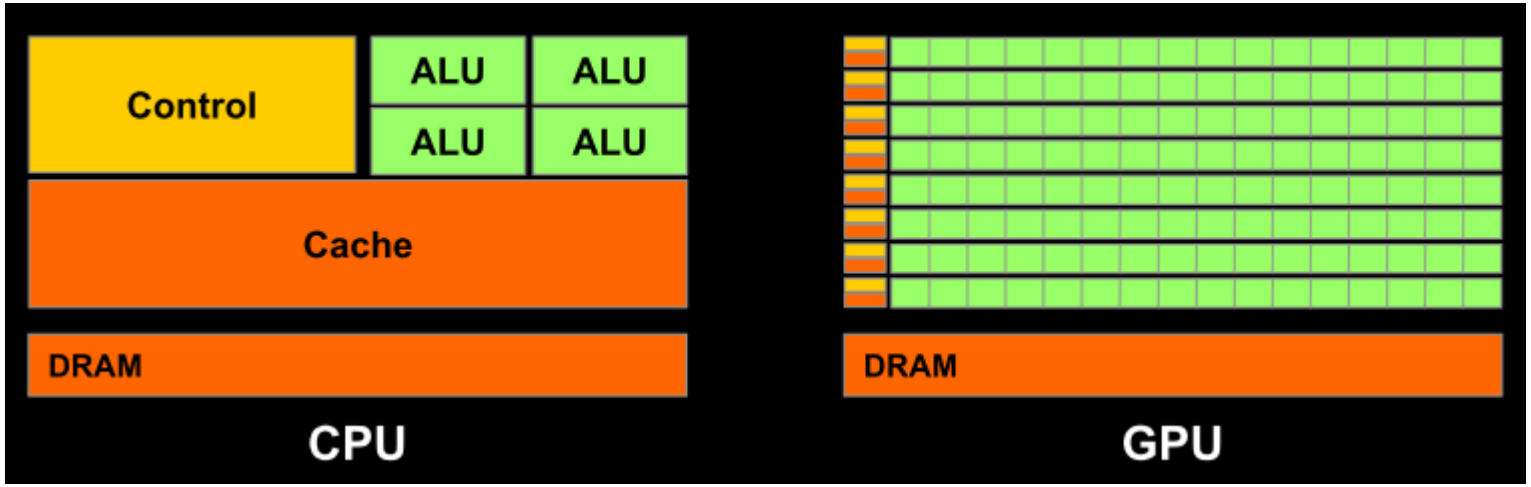
Why?

- Need to understand the difference
  - Latency intolerance versus latency tolerance
  - Task parallelism versus data parallelism
  - Multithreaded cores versus SIMT cores
  - 10s of threads versus thousands of threads

- CPUs: low latency, low throughput
- GPUs: high latency, high throughput
  - GPUs are designed for tasks that tolerate latency

What About Caches?

- GPUs can have more ALUs in the same area and therefore run more threads of computation



GPU Future

- High-end desktops have separate GPU chip, but trend towards integrating GPU on same die as CPU (already in laptops, tablets and smartphones)
  - Advantage is shared memory with CPU, no need to transfer data
  - Disadvantage is reduced memory bandwidth compared to dedicated smaller-capacity specialized memory system
    - Graphics DRAM (GDDR) versus regular DRAM (DDR3)
- Will GP-GPU survive? Or will improvements in CPU DLP make GP-GPU redundant?
  - On same die, CPU and GPU should have same memory bandwidth
  - GPU might have more FLOPS as needed for graphics anyway

Acknowledgements

- These slides contain material developed and copyright by:
  - Krste Asanovic (UCB)
  - Mohamed Zahran (NYU)

- “An introduction to modern GPU architecture”. Ashu Rege. NVIDIA.