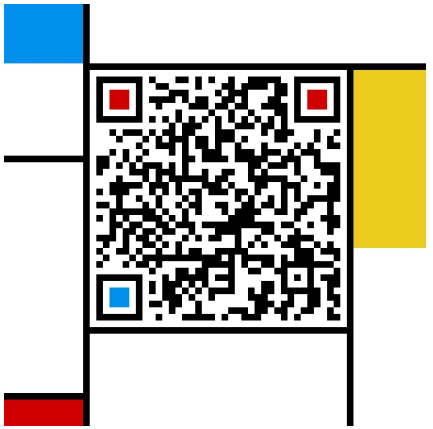
计算机组成与系统结构
Computer Organization & System Architecture

Huang Kejie (黄科杰) 百人计划研究员
Office: 玉泉校区老生仪楼 304
Email address: huangkejie@zju.edu.cn
HP: 17706443800



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

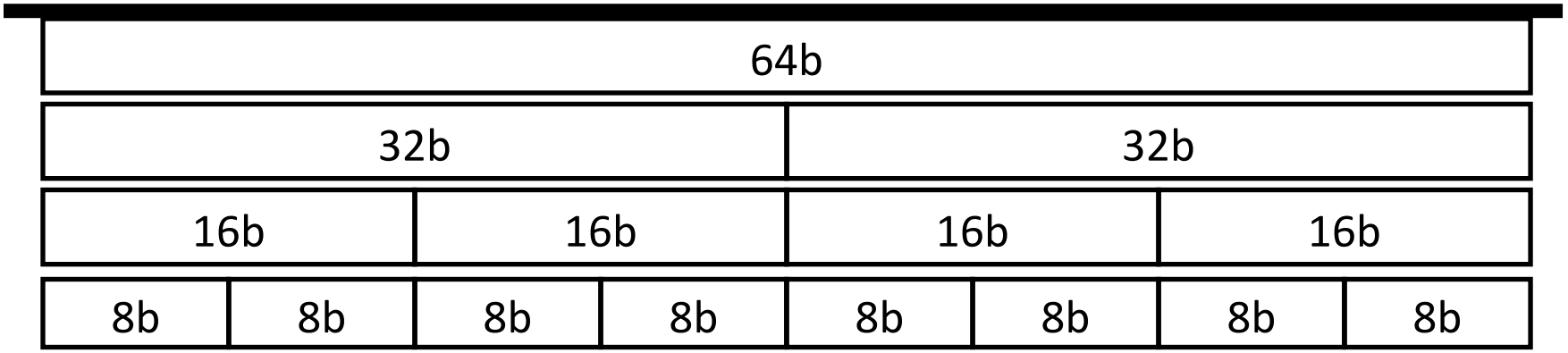
Resurgence of DLP

- Convergence of application demands and technology constraints drives architecture choice

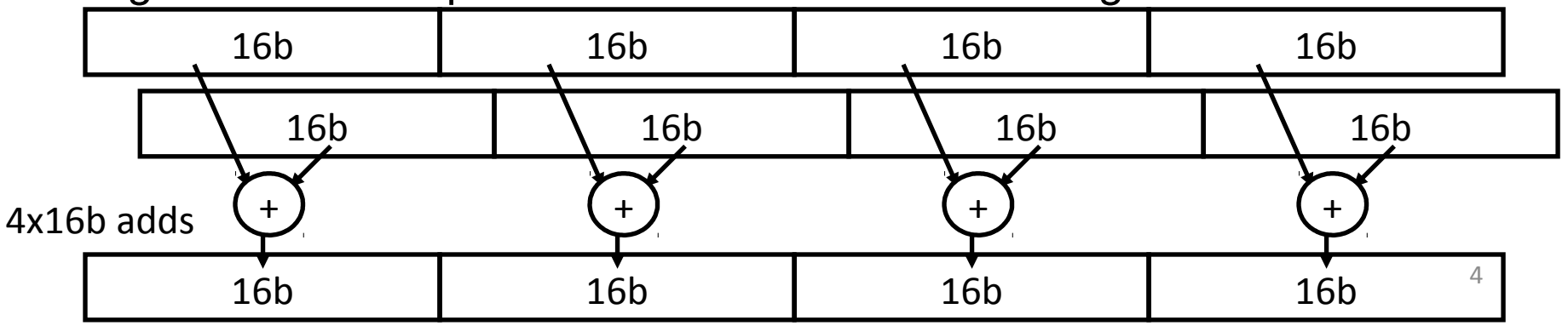
- New applications, such as graphics, machine vision, speech recognition, machine learning, etc. all require large numerical computations that are often trivially data parallel

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

Packed SIMD Extensions



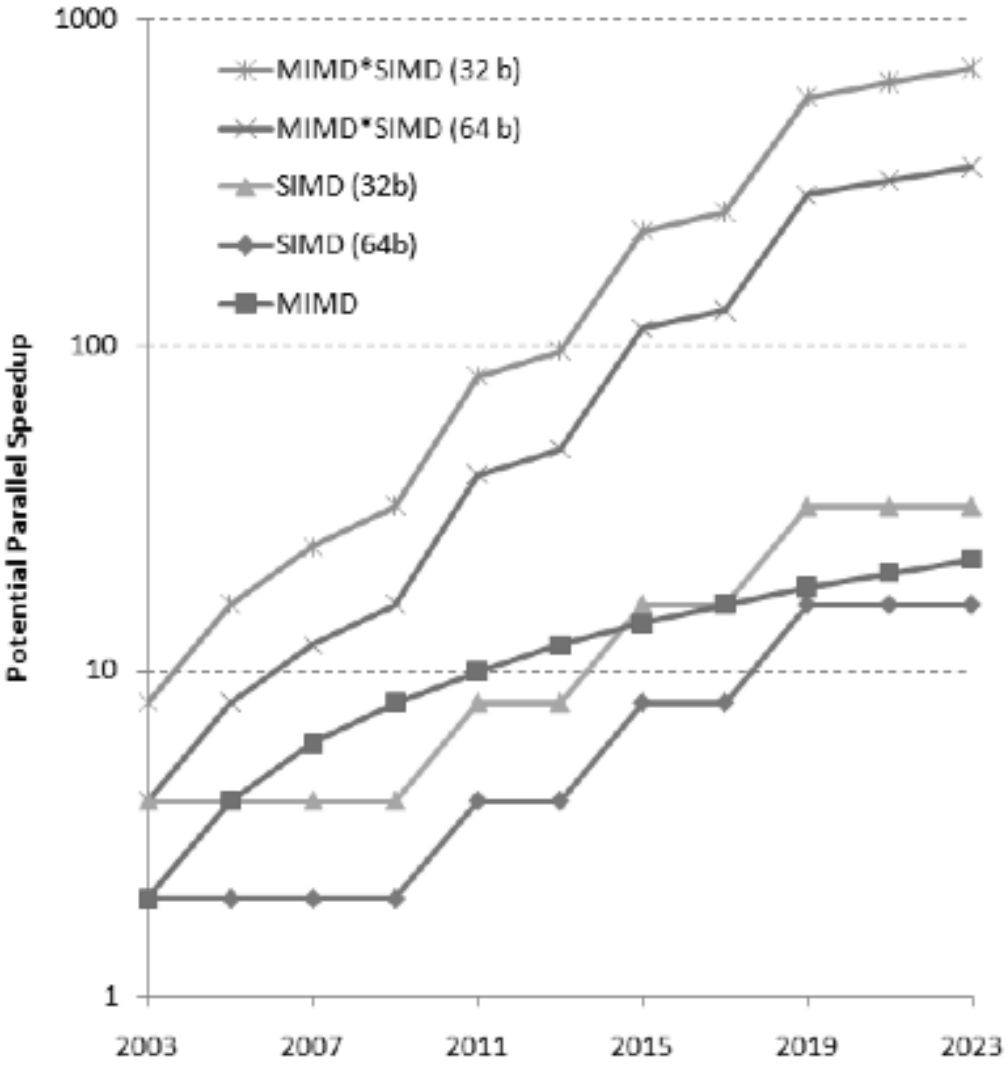
- Very short vectors added to existing ISAs for microprocessors
- Use existing 64-bit registers split into 2x32b or 4x16b or 8x8b
  - Lincoln Labs TX-2 from 1957 had 36b datapath split into 2x18b or 4x9b
  - Newer designs have wider registers
    - 128b for PowerPC Altivec, Intel SSE2/3/4
    - 256b/512b 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) , adding scatter/gather

DLP important for conventional CPUs



- 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)

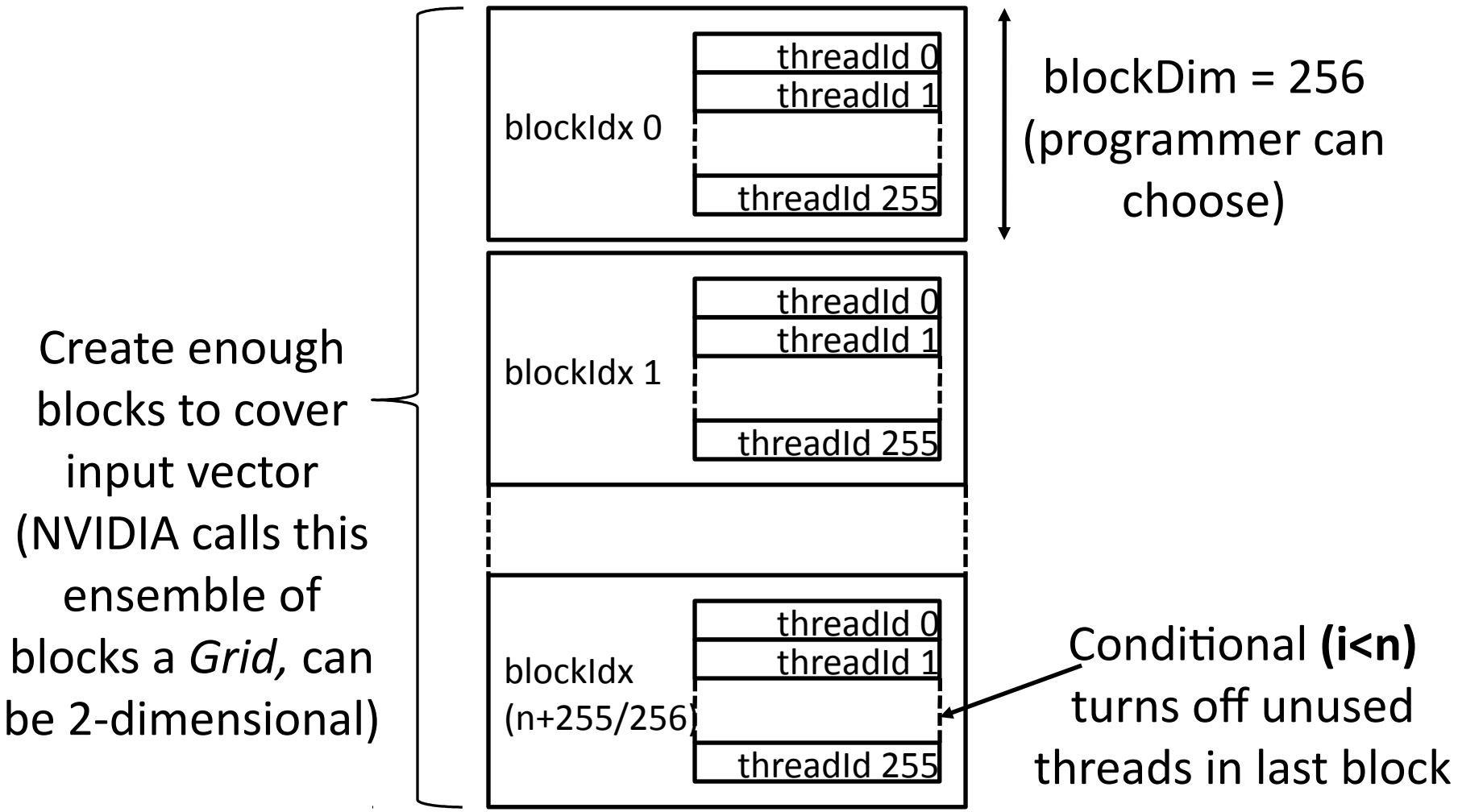
- In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language: CUDA
  - "Compute Unified Device Architecture"
  - Subsequently, broader industry pushing for OpenCL, a vendor-neutral version of same ideas.
- Idea: Take advantage of GPU computational performance and memory bandwidth 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

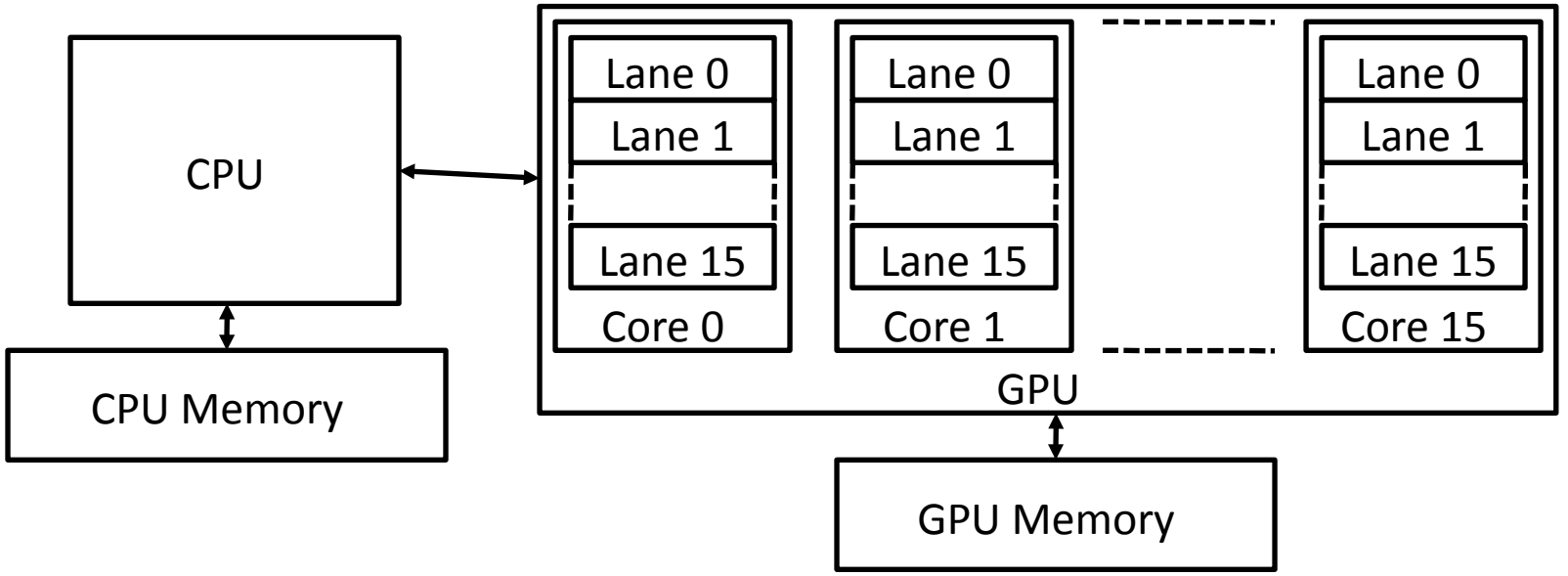
- Computation performed by a very large number of independent small scalar threads (CUDA threads or microthreads) grouped into thread blocks.

```
// 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



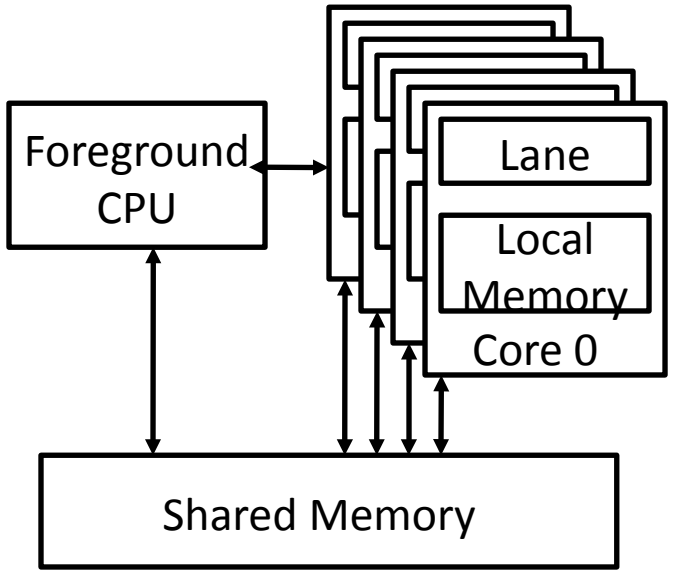
Hardware Execution Model

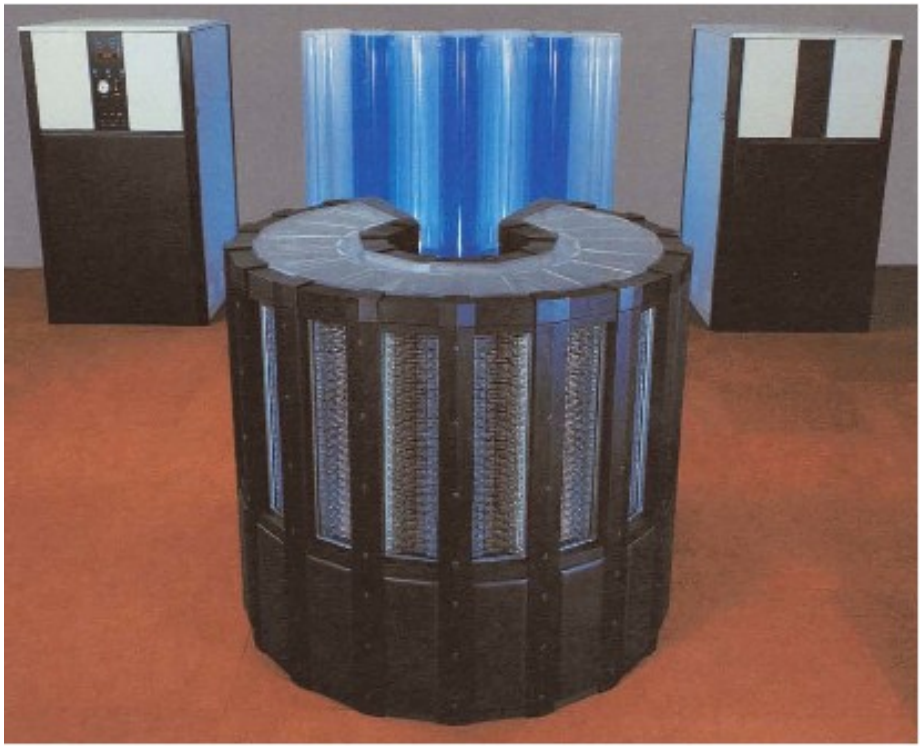


- GPU is built from multiple parallel cores, each core contains a multithreaded SIMD processor with multiple lanes but with no scalar processor
  - some adding “scalar coprocessors” now
- CPU sends whole “grid” over to GPU, which distributes thread blocks among cores (each thread block executes on one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

Historical Retrospective, Cray-2 (1985)

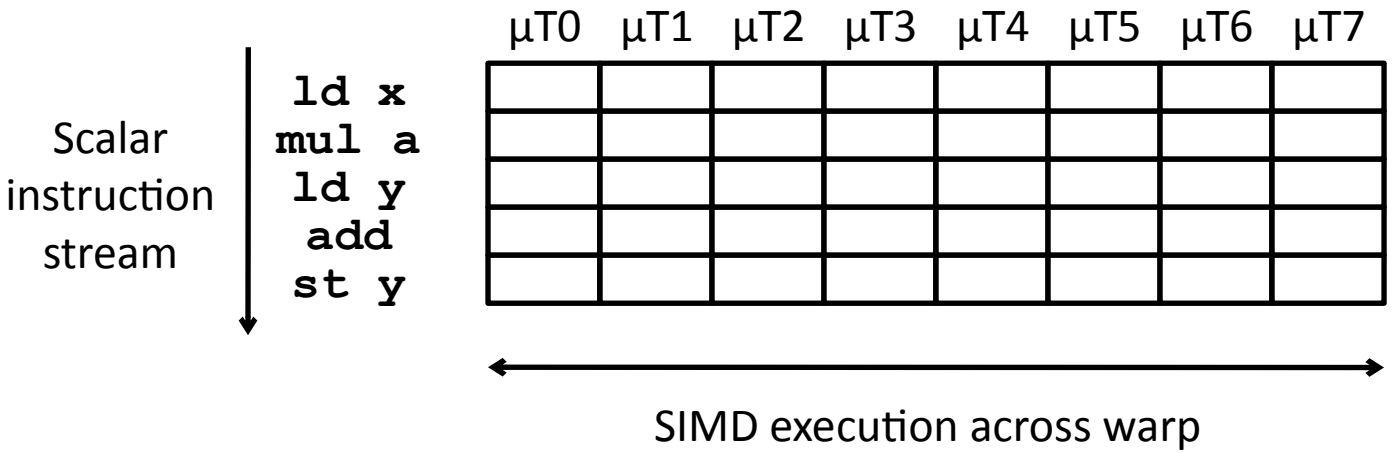
- 243MHz ECL logic
- 2GB DRAM main memory (128 banks of 16MB each)
  - Bank busy time 57 clocks!
- Local memory of 128KB/core
- 1 foreground + 4 background vector processors





“Single Instruction, Multiple Thread” (SIMT)

- GPUs use a SIMT model, where individual scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware (NVIDIA groups 32 CUDA threads into a warp)



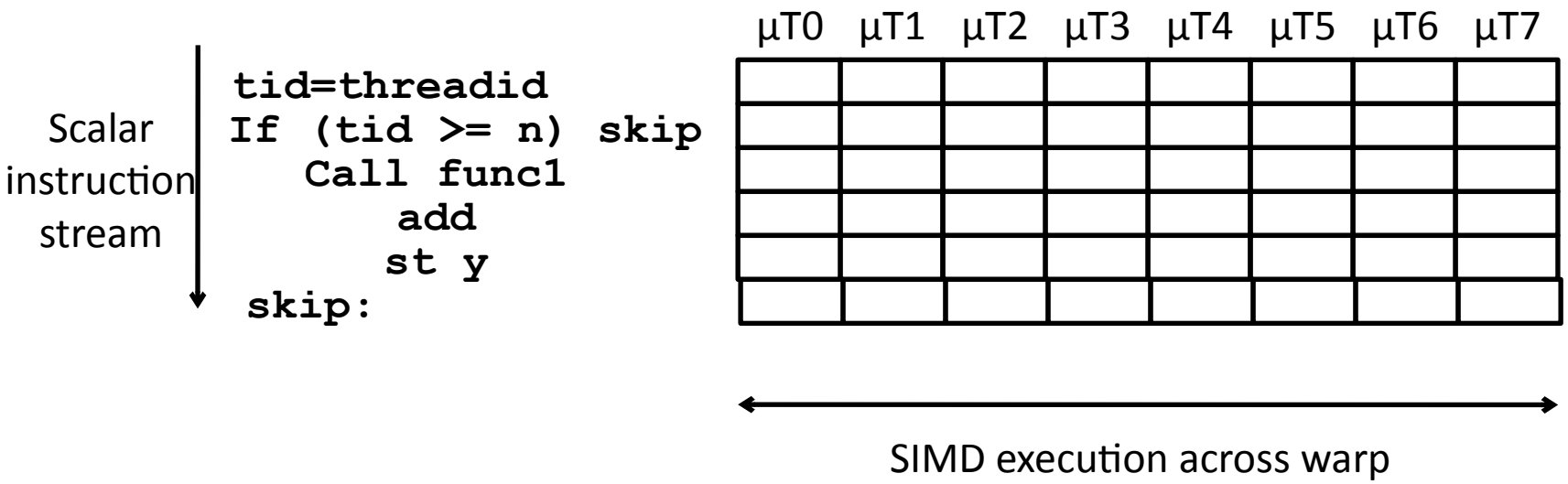
Implications of SIMT Model

- 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?

NVIDIA Instruction Set Arch.

- ISA is an abstraction of the hardware instruction set
  - "Parallel Thread Execution (PTX)"
    - opcode.type d,a,b,c;
  - Uses virtual registers
  - Translation to machine code is performed in software
  - Example:

```
shl.s32 R8, blockIdx, 9    ; Thread Block ID * Block size (512 or 29)
add.s32      R8, R8, threadIdx       ; R8 = i = my CUDA thread ID
ld.global.f64     RD0, [X+R8]      ; RD0 = X[i]
ld.global.f64     RD2, [Y+R8]      ; RD2 = Y[i]
mul.f64 R0D, RD0, RD4; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2 ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0     ; Y[i] = sum (X[i]*a + Y[i])
```

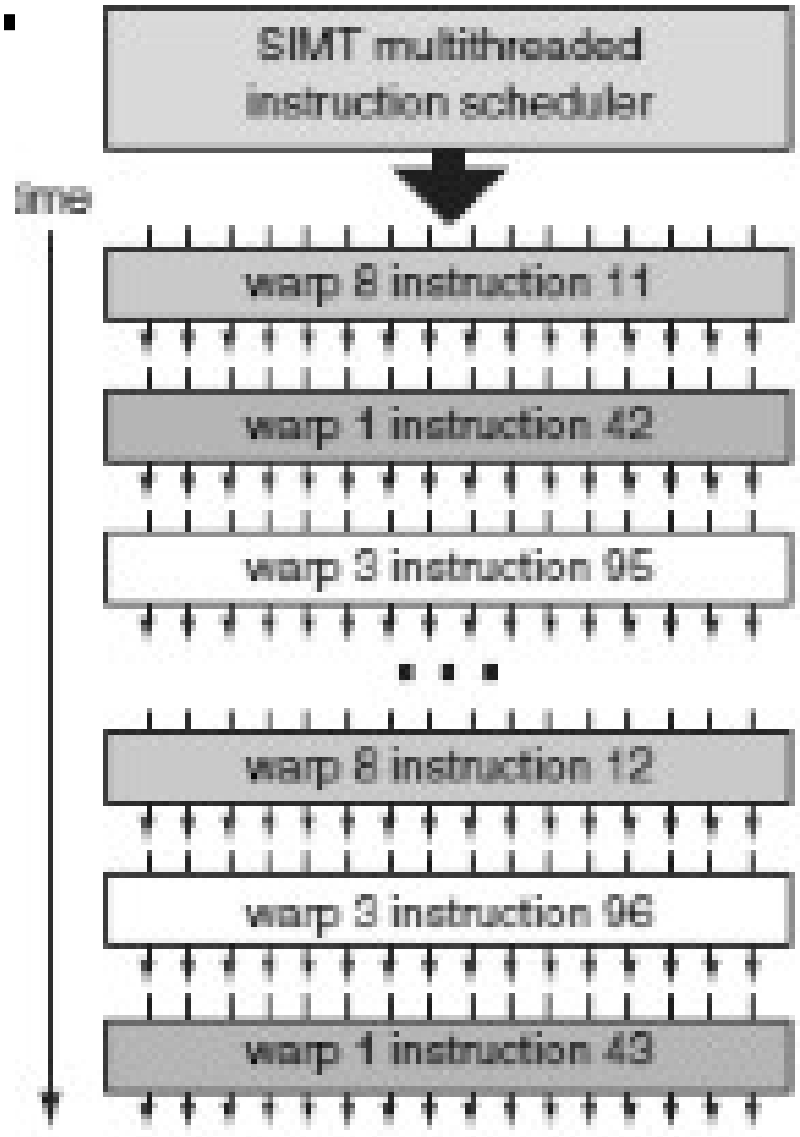
Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane
    - I.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - …and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

Example

```
        if (X[i] != 0)
              X[i] = X[i] – Y[i];
        else X[i] = Z[i];
        ld.global.f64    RD0, [X+R8]  ; RD0 = X[i]
        setp.neq.s32     P1, RD0, #0    ; P1 is predicate register 1
        @!P1, bra        ELSE1, *Push; Push old mask, set new mask bits
        ; if P1 false, go to ELSE1
        ld.global.f64    RD2, [Y+R8]  ; RD2 = Y[i]
        sub.f64    RD0, RD0, RD2            ; Difference in RD0
        st.global.f64    [X+R8], RD0  ; X[i] = RD0
        @P1, bra ENDIF1, *Comp            ; complement mask bits
                                  ; if P1 true, go to ENDIF1
ELSE1:          ld.global.f64 RD0, [Z+R8]     ; RD0 = Z[i]
              st.global.f64 [X+R8], RD0     ; X[i] = RD0
ENDIF1: <next instruction>, *Pop       ; pop to restore old mask
```

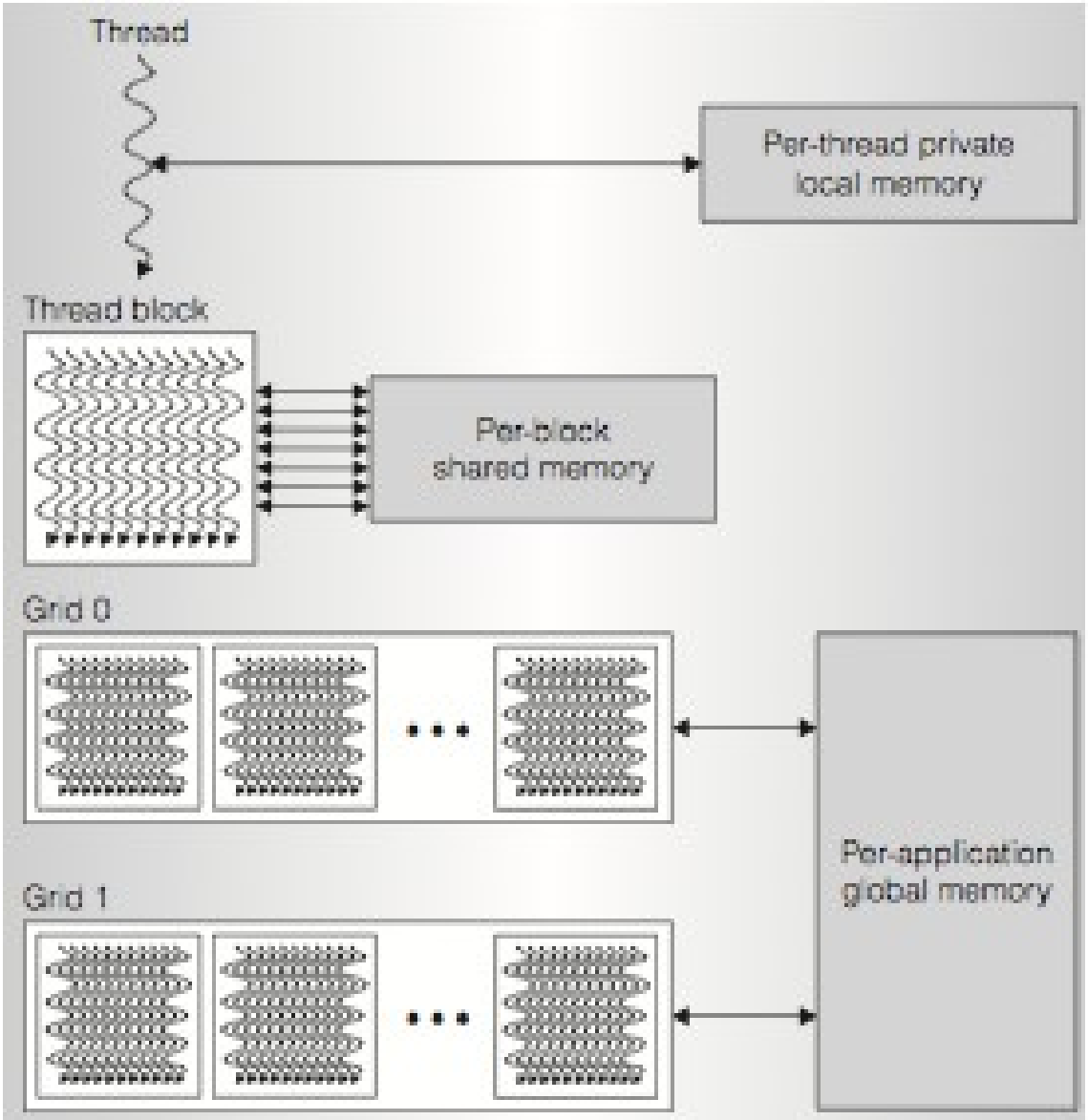
Warps are Multithreaded on Core



[Nvidia, 2010]

- 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

GPU Memory Hierarchy



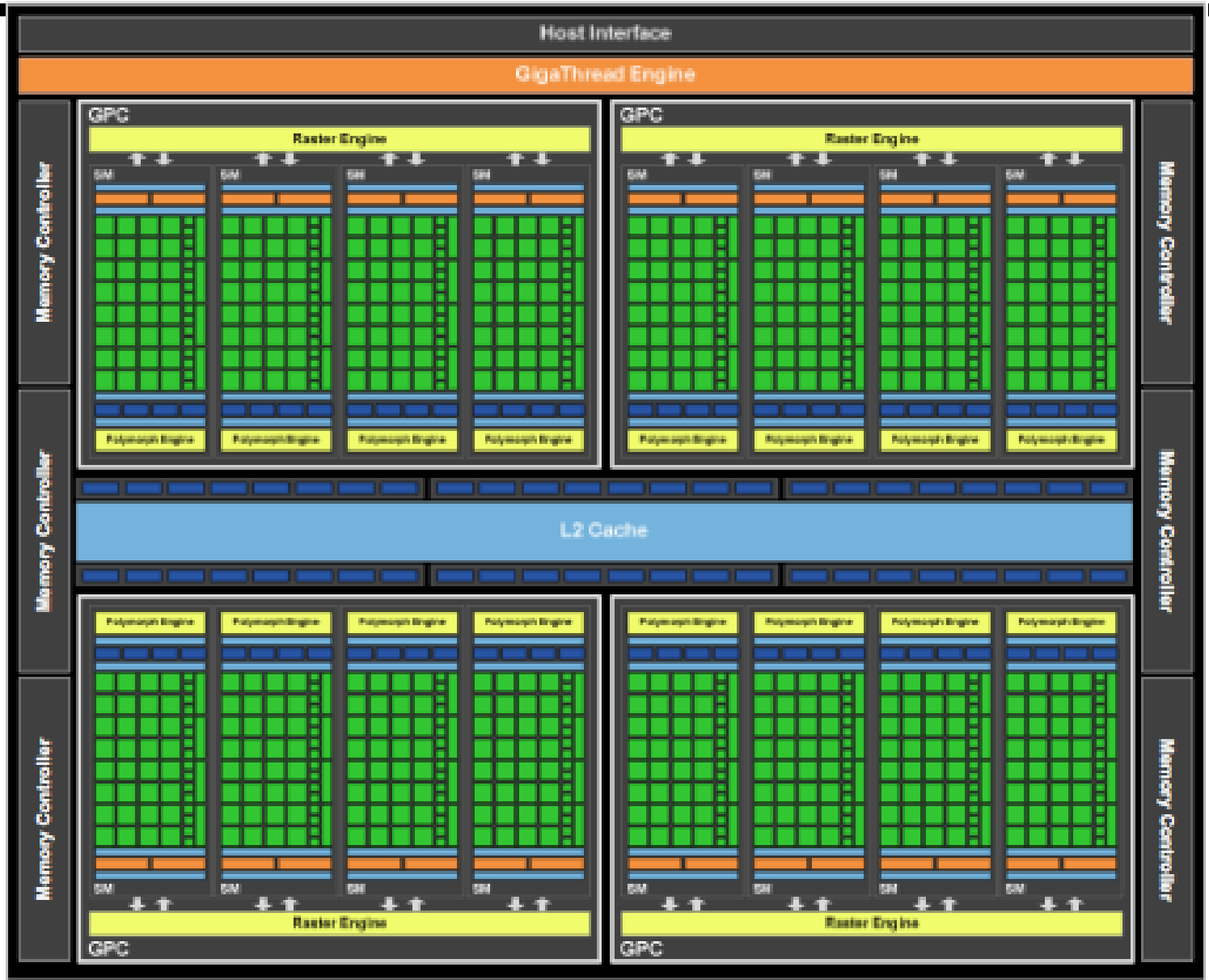
[Nvidia, 2010]

SIMT

- Illusion of many independent threads
- 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

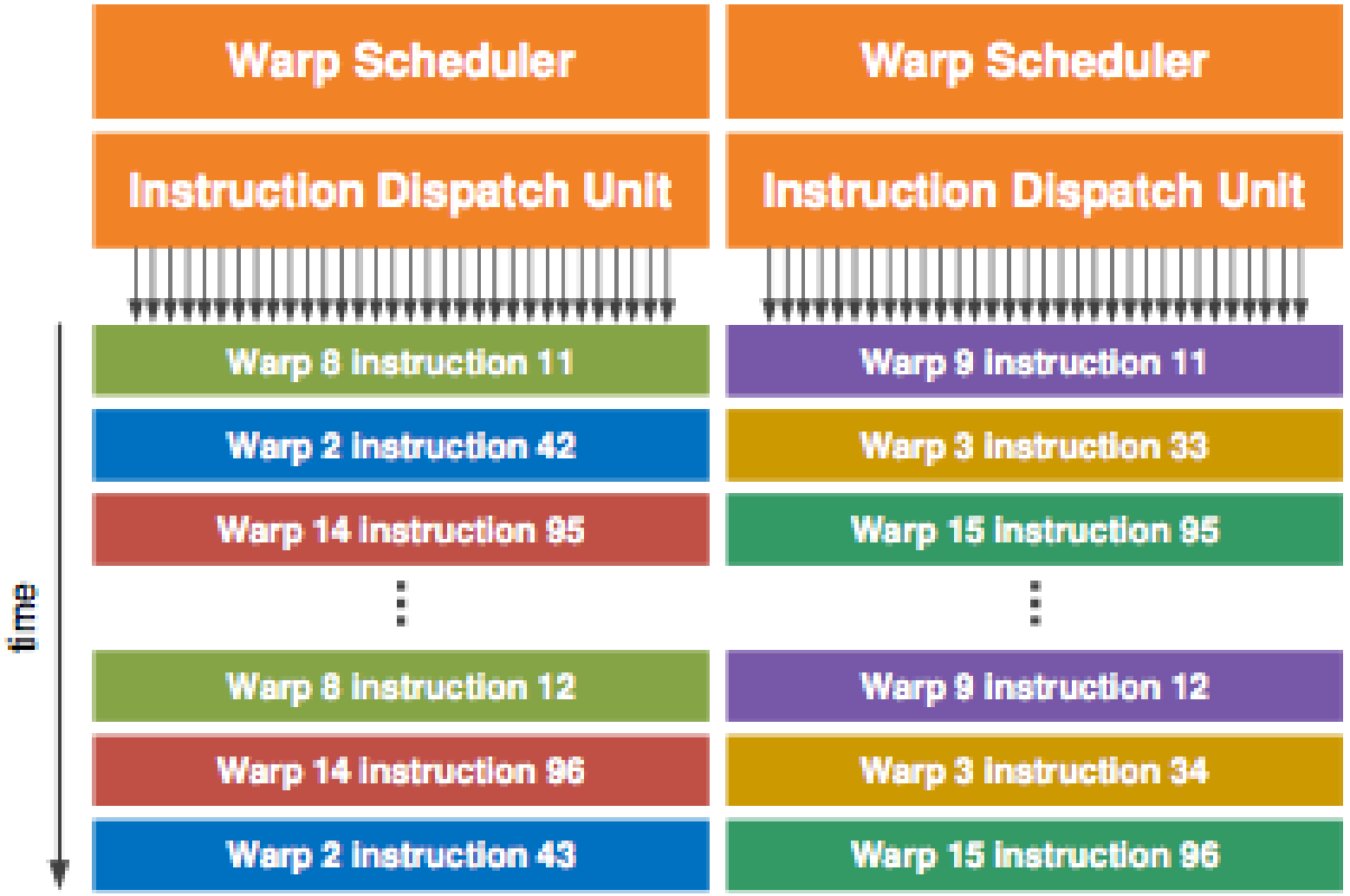
[Nvidia, 2010]



Fermi “Streaming Multiprocessor” Core

NVIDIA Pascal Multithreaded GPU Core

Fermi Dual-Issue Warp Scheduler

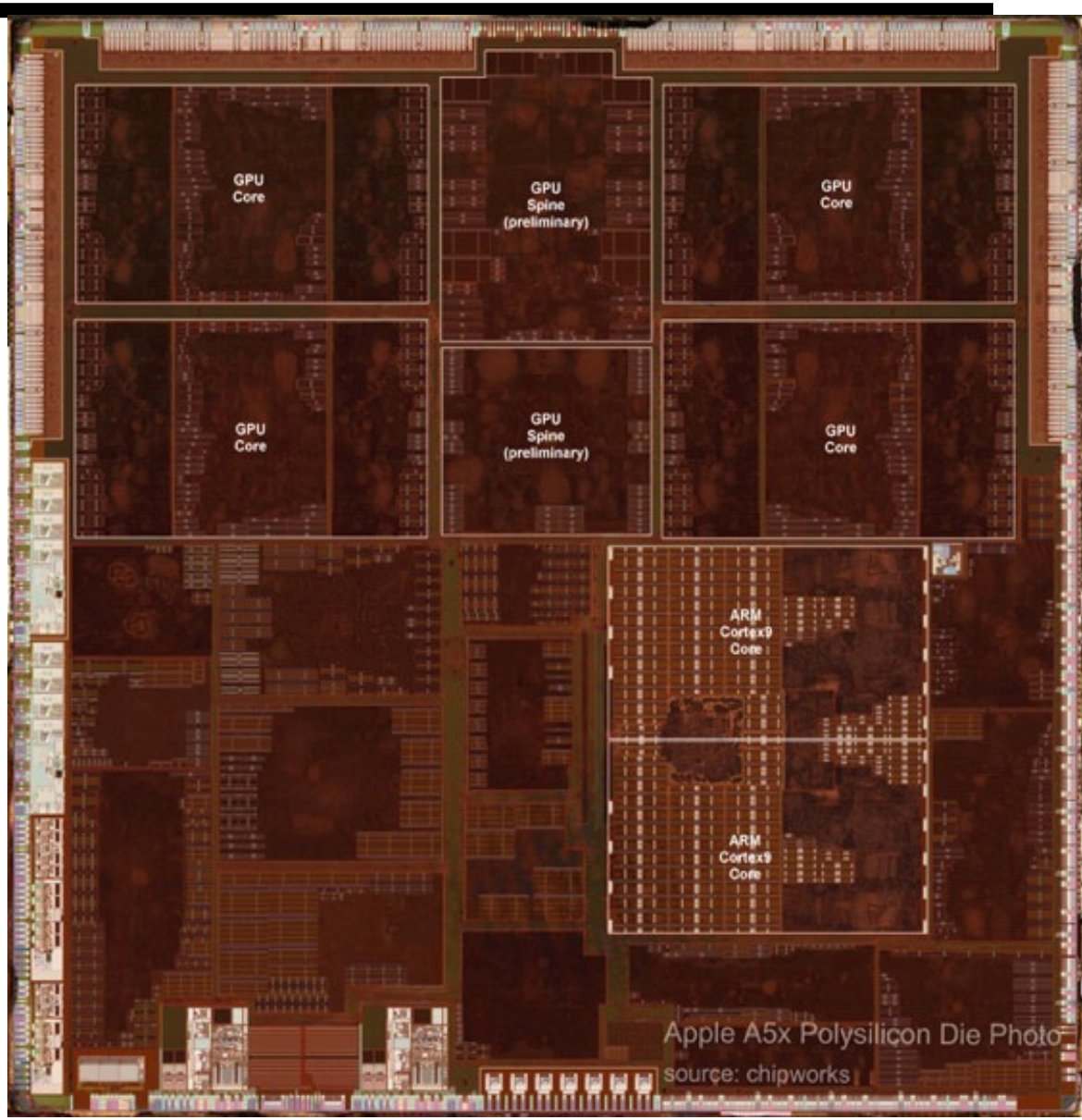


Important of Machine Learning for GPUs

NVIDIA stock price 20x in 5 years

Apple A5X Processor for iPad v3 (2012)

- 12.90mm x 12.79mm
- 45nm technology



[Source: Chipworks, 2012]