Handout 6 GPU

Outline

- **Introduction of HSA**
- **Execution model**
- **Multi-threaded SIMD processor**
- **Fermi Architecture**
- **SIMT ISA: PTX**
- **Memory Hierarchy**
- **Summary on GPGPU**

Heterogeneous System Architecture

- **HSA Foundation**
- **HSA System Architecture Specification**
  - **Version 1.0 Provisional, Released April 2014 (V 1.0,March 16, 2015)**
  - **Define how the hardware operates**
- **HSA Programmers Reference Specification**
  - **Tool chain, software ecosystem**
  - **Define the HSAIL language and object format**
- **HSA Runtime Software Specification**
  - **Define the APIs through which an HSA application uses the platform**

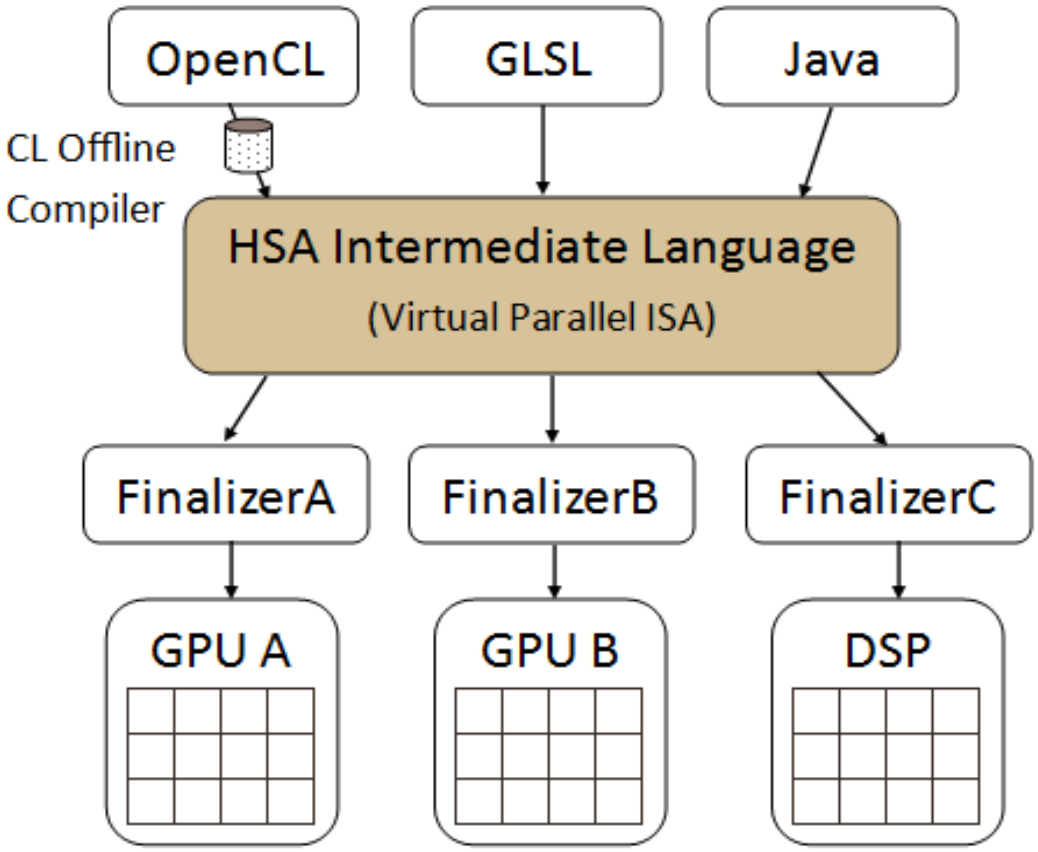
HSA-Aware SIMT GPU

ARM: Mali (Shader core)
Imagination: PowerVR
Nvidia: Tegra (Keplar)

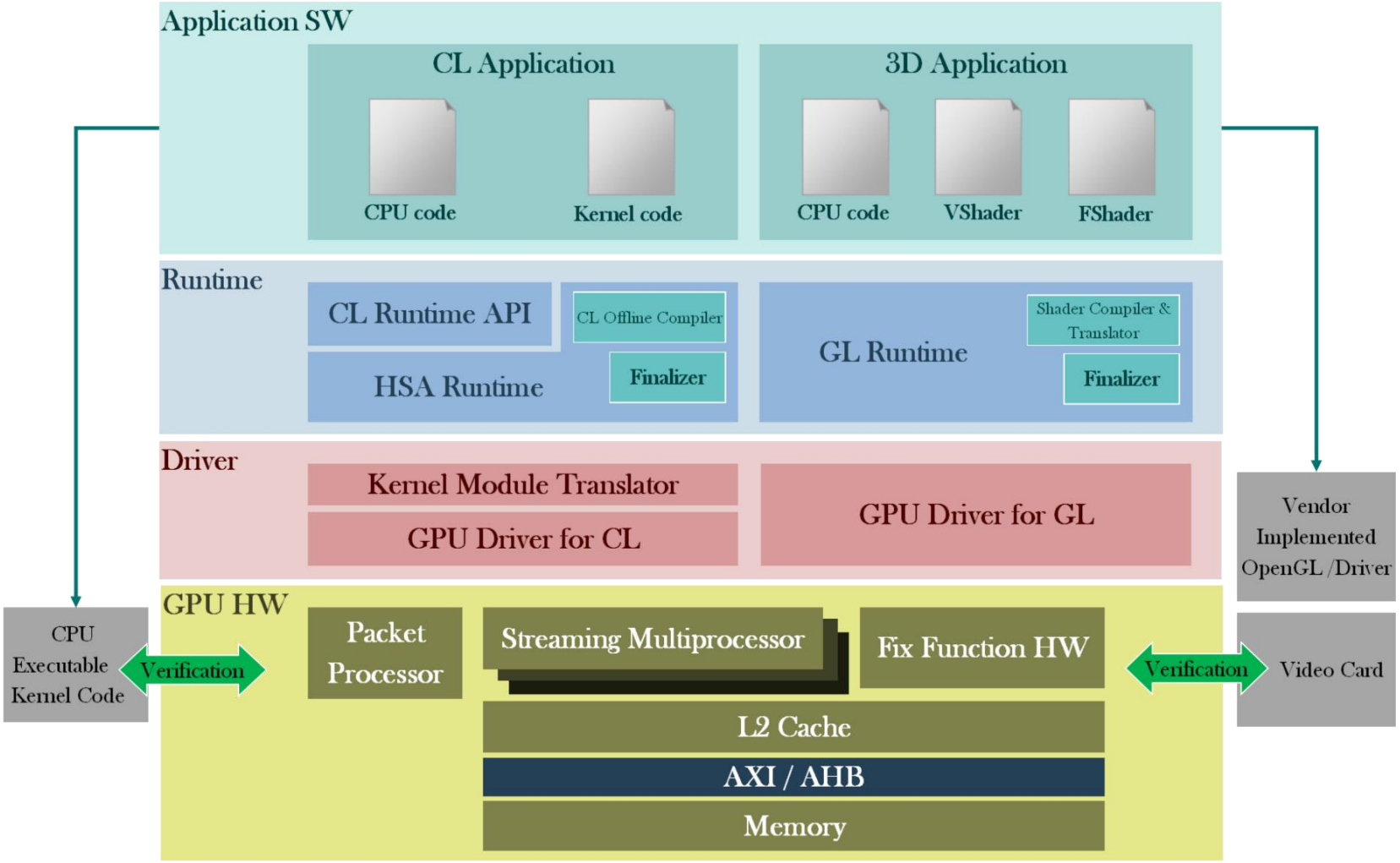
國外技術研發現況分析

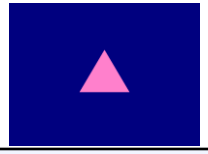
Opportunity:
For computing and graphics:
A GPU of our own binary.
ISA license-free. 自主技術.

Application:
Cloud center, data center,
energy efficient servers,
mobile, machine learning etc.



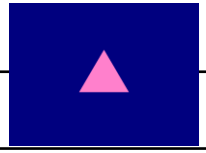
支援OpenCL 及OpenGL 的HSA 相容GPU 系統

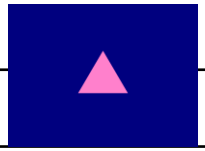


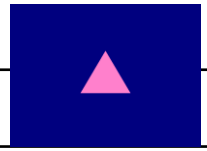


Drawn by Our GPUs

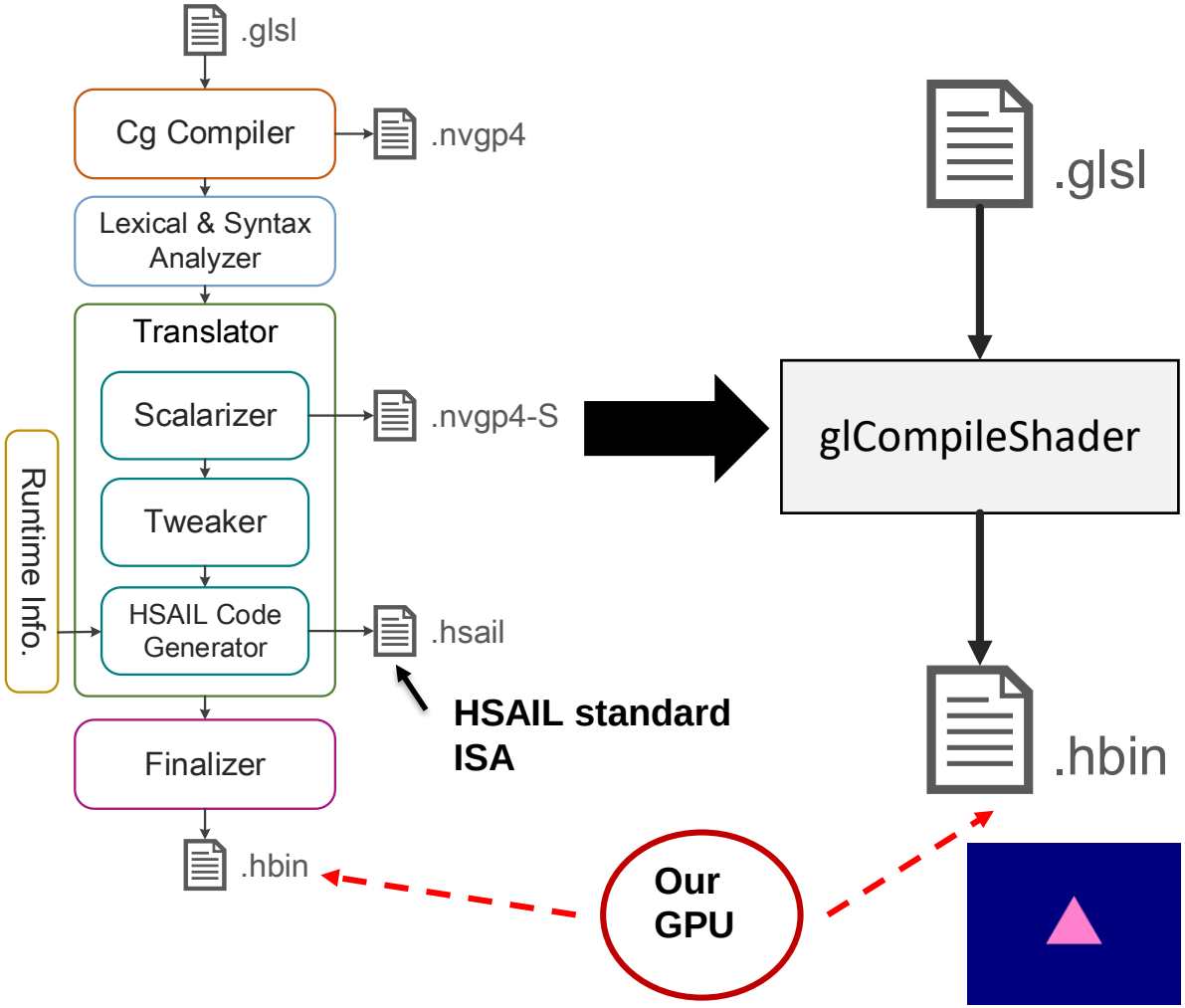
Platform	OpenGLES-sim	OpenGLES-sim Scalar	HSA Simulator Graphic Version
Devel. time	2013-2014	2014-2015	2015/April
Computing model	SIMT Vector (4 lanes, vector size 4)	SIMT Scalar (16 scalars)	SIMT Scalar
Purpose	Graphic	Graphic	Computing & Graphic
Program source	GLSL	GLSL	cl & GLSL
Compiler	cgc	cgc w/ scalarizer	cgc w/ translator CLOC
ISA	nvgp4 (pseudo asm.)	nvgp4-S (pseudo asm.)	**HSAIL Custom binary ISA**
Data Load/Store	Fixed hardware	Fixed hardware	Instruction
Core utilization	Medium	High	High
Warp scheduling	None	None	Yes
SM Cluster	None	None	Yes







API hides all compilation and translation details



NVIDIA cgc is our front end compiler.

OpenGL Runtime APIs

- **OpenGL 2.0. (1)設定GPU 內fix function unit 的configuration**
  (2)透過GL 的API 去引入fragment/vertex shader kernel

glActiveTexture	glAttachShader	glBindTexture	glClear
glClearColor	glClearDepthf	glCompileShader	glCreateProgram
glCreateShader	glCullFace	glDeleteProgram	glDeleteShader
glDeleteTextures	glDepthRangef	glDetachShader	glDisable
glDrawArrays	glDrawElements	glEnable	glEnableVertexAttribArray
glGenerateMipmap	glGenTextures	glGetAttribLocation	glGetError
glGetProgramiv	glGetShaderiv	glGetUniformLocation	glIsProgram
glIsShader	glIsTexture	glLinkProgram	glShaderBinary
glShaderSource	glTexImage2D	glTexParameteri	glUniform1f
glUniform1i	glUniform1fv	glUniform1iv	glUniform2f
glUniform2i	glUniform2fv	glUniform2iv	glUniform3f
glUniform3i	glUniform3fv	glUniform3iv	glUniform4f
glUniform4i	glUniform4fv	glUniform4iv	glUniformMatrix2fv
glUniformMatrix3fv	glUniformMatrix4fv	glUseProgram	glValidateProgram
glVertexAttribPointer	glViewport		

OpenCL & HSA Runtime APIs

- **OpenCL 1.2 and HSA 1.0**

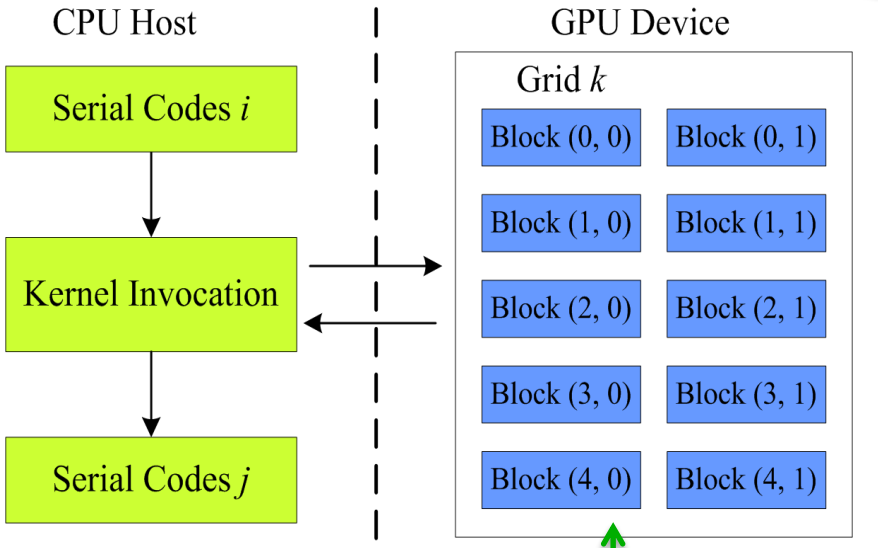
clBuildProgram	clCreateBuffer	clCreateCommandQueue
clCreateCommandQueueWithProperties	clCreateContext	clCreateContextFromType
clCreateKernel	clCreateProgramWithSource	clEnqueueMapBuffer
clEnqueueNDRangeKernel	clEnqueueReadBuffer	clEnqueueUnmapMemObject
clEnqueueWriteBuffer	clFinish	clFlush
clGetContextInfo	clGetDeviceIDs	clGetDeviceInfo
clGetEventInfo	clGetEventProfilingInfo	clGetKernelWorkGroupInfo
clGetPlatformIDs	clGetPlatformInfo	clGetProgramBuildInfo
clGetProgramInfo	clReleaseCommandQueue	clReleaseContext
clReleaseKernel	clReleaseMemObject	clReleaseProgram
clSetKernelArg		

Outline

- Introduction
- Execution model
- Multi-threaded SIMD processor
- Fermi Architecture
- SIMT ISA: PTX
- Memory Hierarchy
- Summary on GPGPU

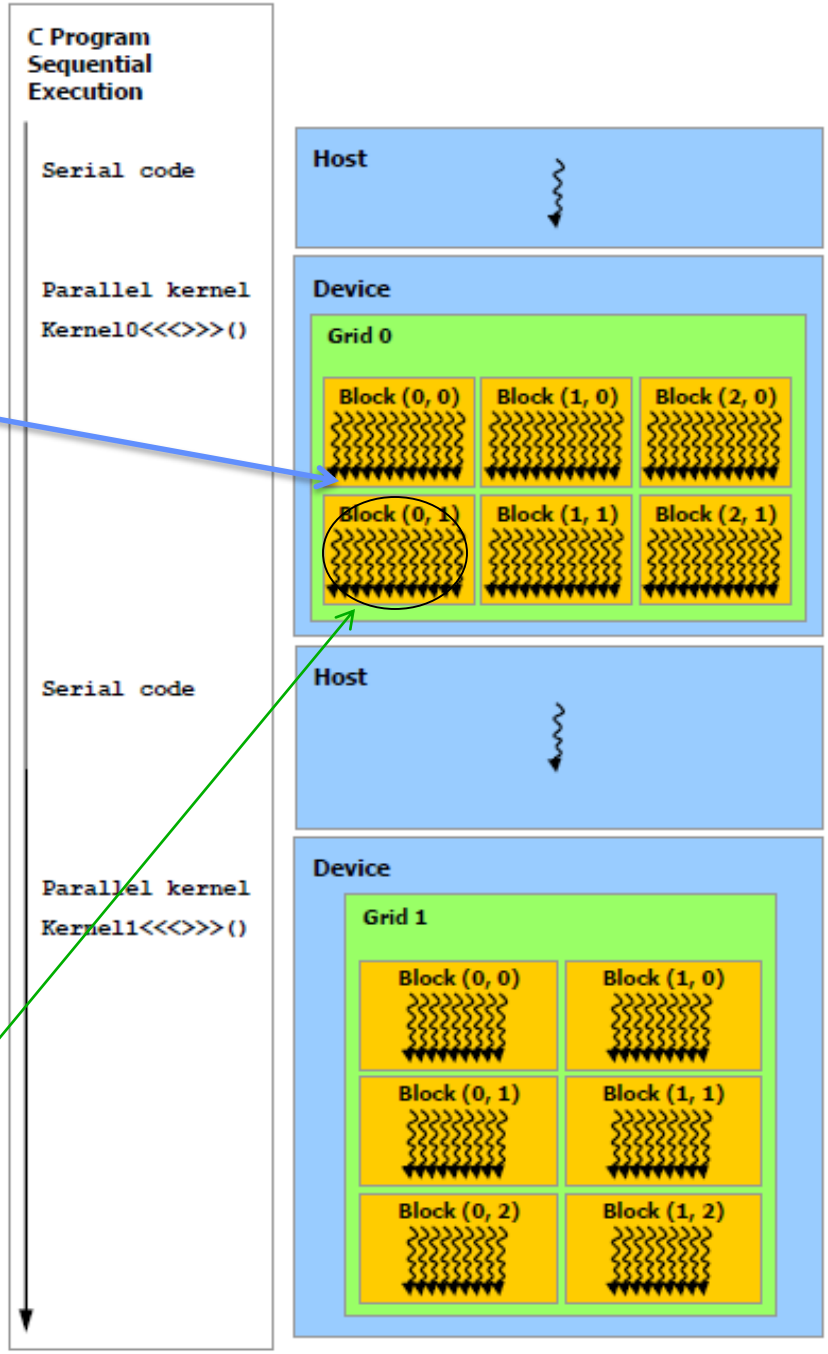
Parallel Kernel

Thread blocks can be executed independently and in any order



2 x 5 blocks

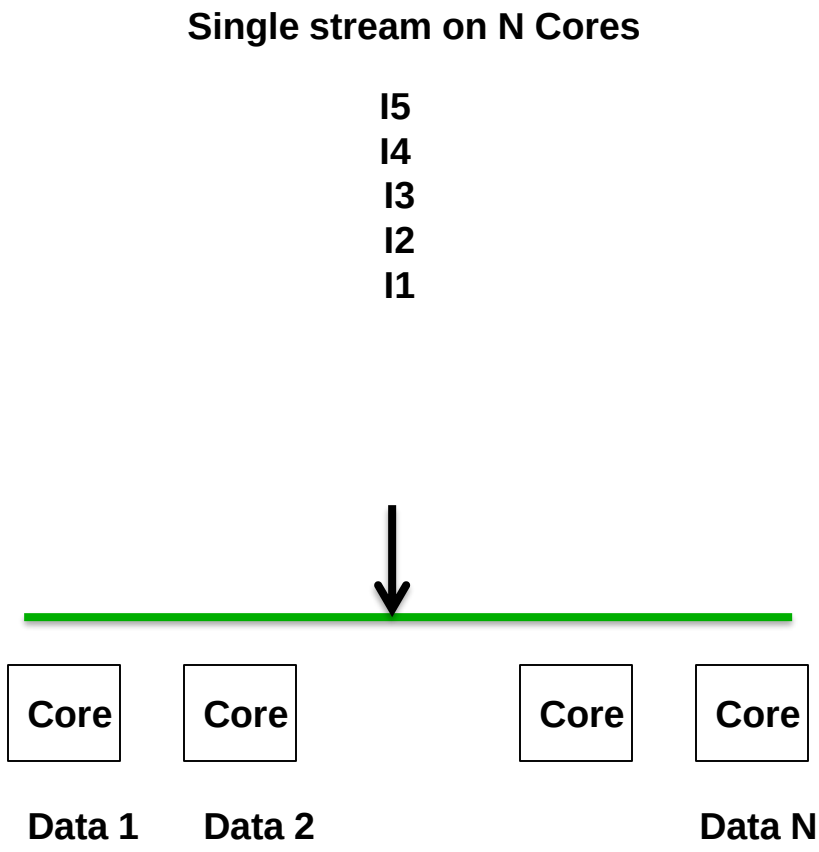
Programmer determines the parallelism by specifying the grid dimensions and the number of threads per SIMD processor



Execution model

Single Instruction Multiple Threading

- **Single Instruction Multi-Threading**
- **Get one instruction and dispatch to every processor units.**
- **Fetch one stream -> N threads (of the same code) in execution**
- **Each thread is independent to each other.**
- **All cores execute instructions in lockstep mode.**



Threads and Blocks

- A thread is associated with each data element
- Threads are organized into blocks
  - A thread block is assigned to a processor called multithreaded SIMD processor
- Blocks are organized into a grid
  - thread blocks can run independently and in any order

- A grid is the code that runs on a GPU that consists of a set of thread blocks.
- GPU hardware handles thread management, not applications or OS

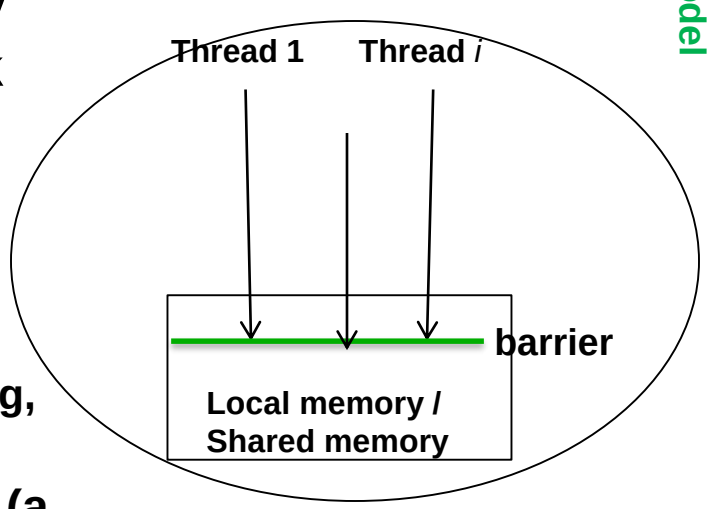
A thread; user defined entity

- A thread within a thread block (group) executes an instance of the kernel (code to execute)
  - Has a thread ID in the group
  - Has its program counter
  - Has its registers, per-thread private memory
    - » For register spills, procedure call (stack)
  - Use off-chip DRAM for private memory
    - » Why? (Data to work on)
  - Can have L1 and L2 cache to cache private memory
  - Map onto a SIMD lane
  - SIMD lanes do not share private memories

A thread is an instance of program code in execution!

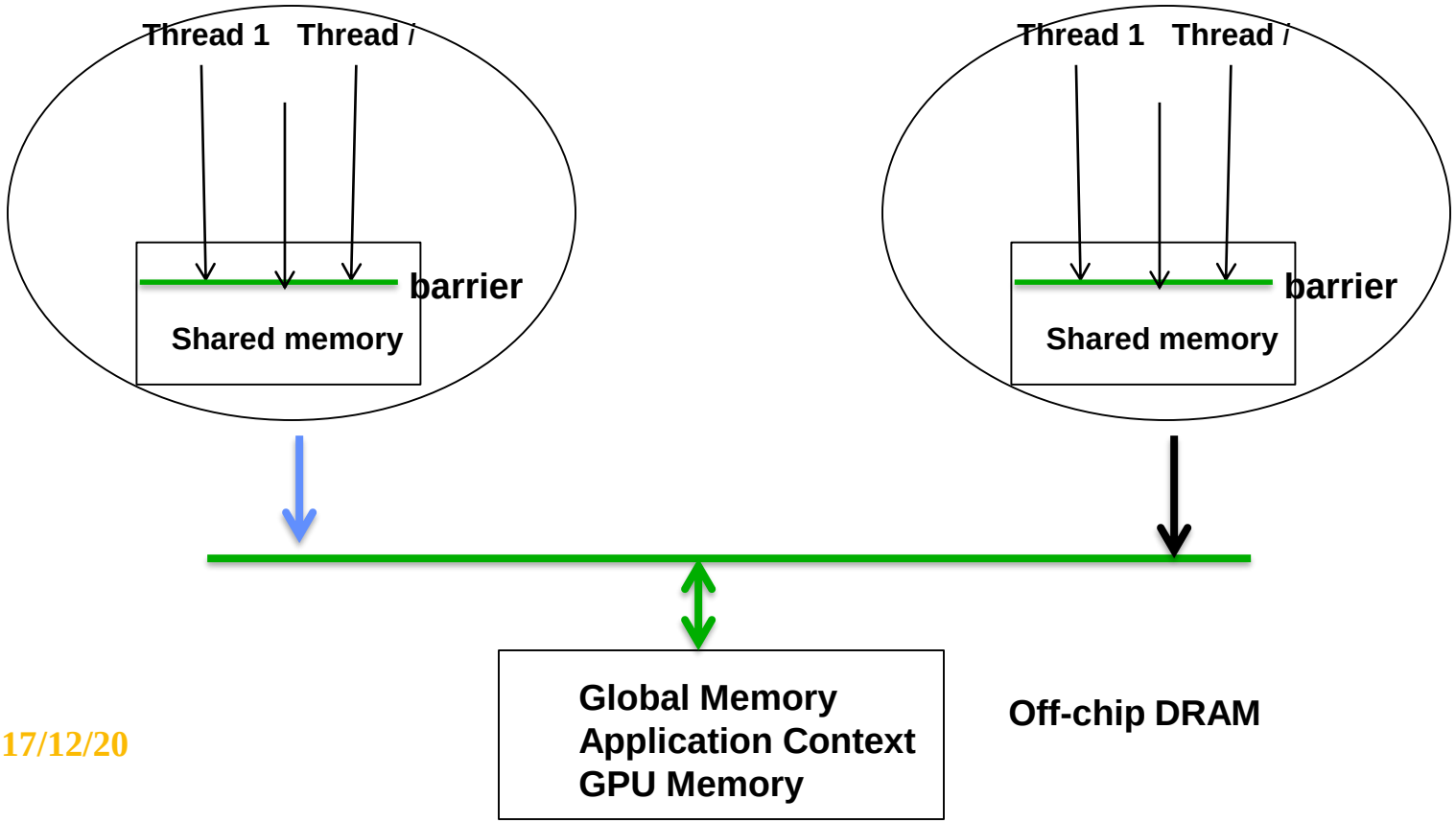
A group of threads: thread block

- **A thread block: a group of concurrently executing threads within a thread block**
  - **Has a thread block ID in a grid**
  - **Synchronization through barrier**
  - **And communicate through a block level shared memory**
    - » **Inter-thread communication, data sharing, result sharing**
  - **Map onto a multithread SIMD processor (a block of several SIMD lanes)**
  - **The SIMD processor dynamically allocates part of the LM to to a thread block when it creates the thread block and frees the memory when all the threads in the thread block exit.**
  - **The local memory is shared by the SIMD lanes within the multithreaded SIMD processor**



A group of thread blocks: grid

- A thread grid: a group of thread blocks that execute the same kernel, read/write inputs/results from/to global memory, synchronize dependent kernel calls through global memory,



Programmer's job

- CUDA programmer explicitly specifies the parallelism
  - Set grid dimensions
  - Number of threads per SIMD processors
- One thread works on one element; no need to synchronize among threads when writing results to memory.

Programming example

- **C code**

```
daxpy (n, 2.0, x, y)  // invoke daxpy
void daxpy ( int n, double a, double *x, double *y)
{
For (int i=0; i<n; ++i)
  y[i] = a*x[i] + y[i];  }  // no loop carried dependence, vectorizable loop
```

- **CUDA_code // launch n threads by invoking DAXPY with 256 threads per thread block**

```
_host_  // declare this is a system processor function
Int nblocks = (n+255)/256;       // how many thread blocks are needed?
daxpy<<<nblocks, 256>>>(n, 2.0, x, y)  //function call: daxpy<<<dimGrid, dimBlock>>>
_device_ // declare this is a GPU function in which all variables are allocated in GPU Mem.
void daxpy ( int n, double a, double *x, double *y)
{ int i = blockIdx.x*blockDim.x + threadIdx.x;  // one thread works on one vector element
// compute element index i based on block ID, the number of threads per block, and the
    thread ID.
  if (i<n) y[i]= a*x[i] + y[i];  // if index i is within the array, then compute y[i]
}
```

Outline

- Introduction
- Execution model
- Multi-threaded SIMD processor
- Fermi Architecture
- SIMT ISA: PTX
- Memory Hierarchy
- Summary on GPGPU

GPU terms-1

Type	More descrip-tive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.

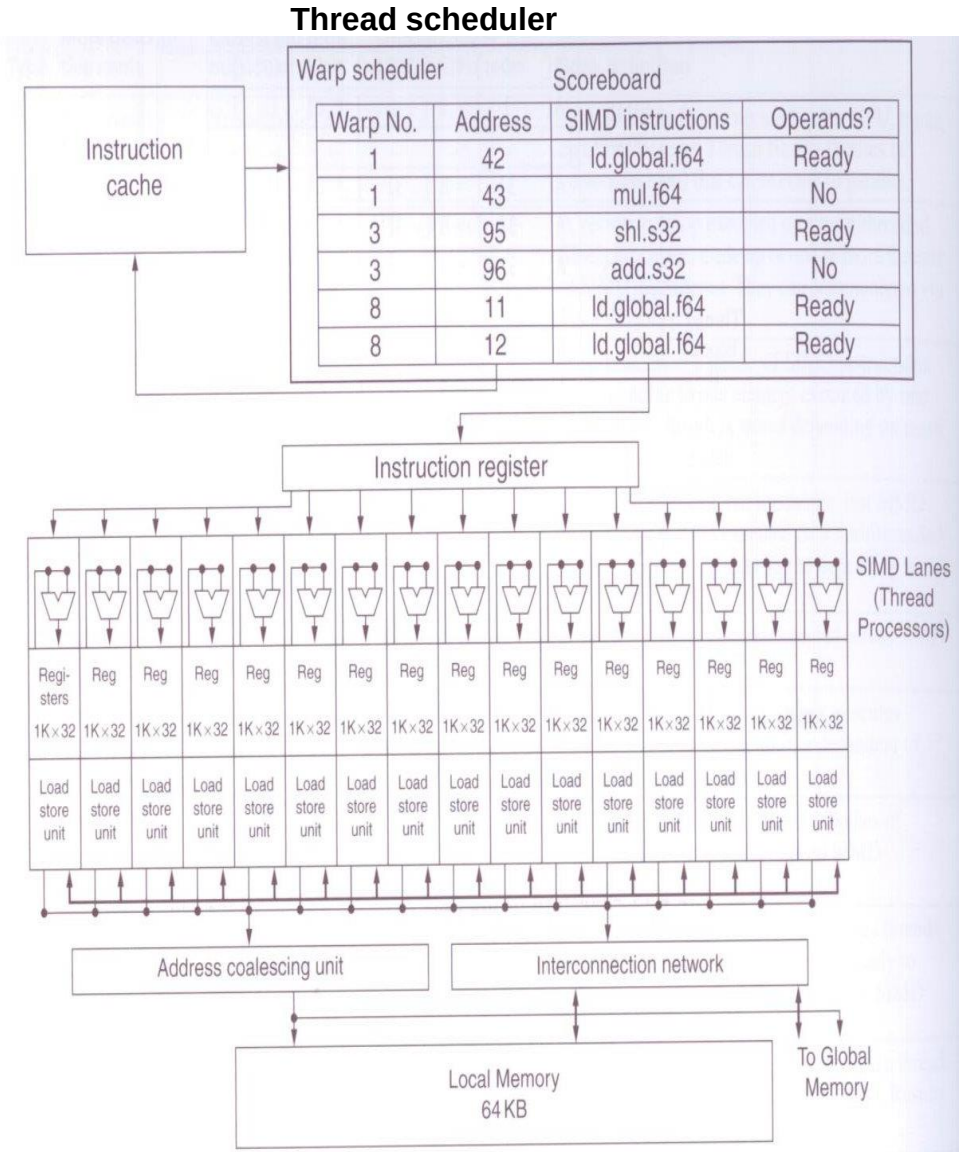
GPU terms-2

		CUDA term	
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors. **Run a thread block**
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

Multi-threaded SIMD processor

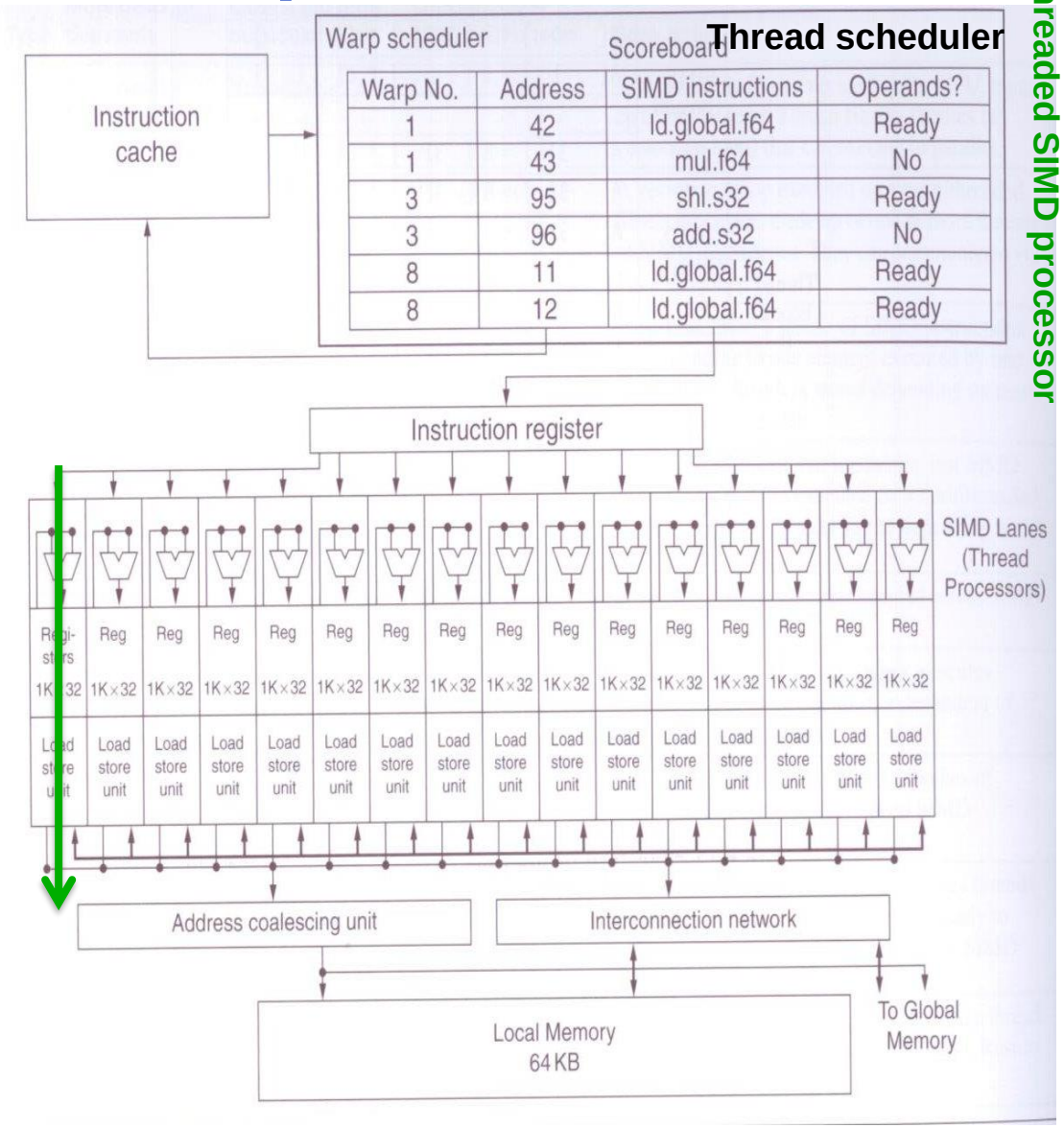
A multithreaded SIMD processor

- This is a multithreaded SIMD processor which runs a thread block
- Thread block scheduler
  - Determine the # of thread blocks required for the task (e.g., vectorizable loop)
  - and keep allocating them to different SIMD processors until the loop is completed.
- Warp scheduler, i.e., thread scheduler
  - Inside the SIMD processor, schedule instructions from ready-to-run threads



A multithreaded SIMD processor-2

- **Scoreboard: keep track of which instruction is ready for execution**
- **Inside a SIMD processor, Warp scheduler, i.e., thread scheduler**
- **A SIMD lane = a thread processor = a CUDA thread, working on one element.**
- **A PTX instruction = A SIMD instruction which is executed across SIMD lanes**

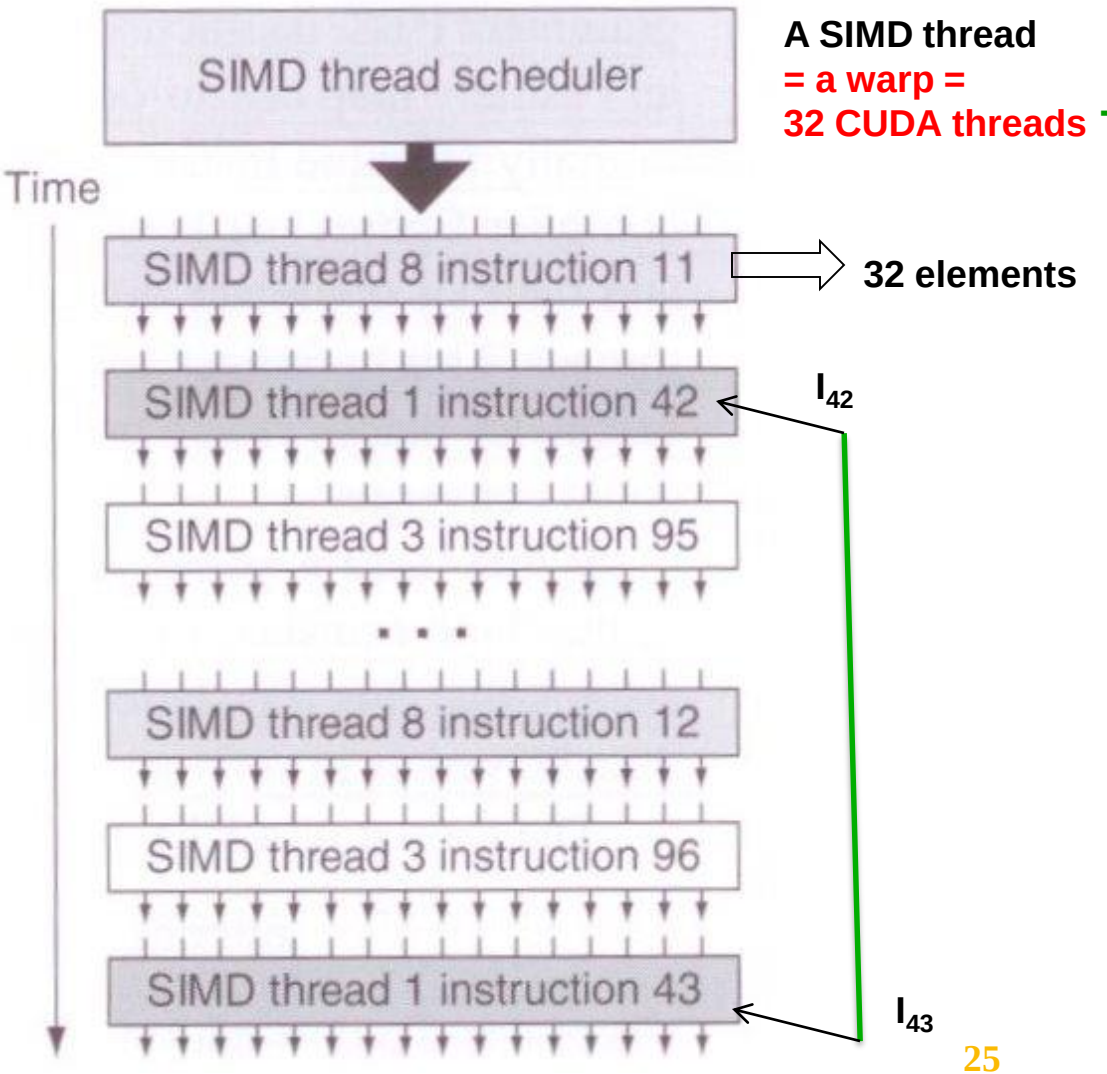


A multithreaded SIMD processor-3

- **Since a SIMD lane = a thread processor = a CUDA thread, working on one element.**
- **So, if a 32-wide thread of SIMD instructions is mapped onto this 16 lanes of thread processors.**
- **Then, this 32-element vector takes 2 clock cycles, i.e. chime = 2 clock cycles**
- **The 16 lane of processors executes this in lock-step and only scheduled at the beginning.**
- **Need not to pick up the NEXT SIMD instruction in the sequence within a thread for scheduling.**

SIMD Instruction scheduling

- **Select a ready thread and issues an instruction synchronously to all the SIMD lanes executing the SIMD thread.**
- **Within a SIMD processor, why scheduling this way?**
  - **Hide memory latency**
  - **Wait for pipeline stalls**
  - **Wait for execution latency**



Outline

- Introduction
- Execution model
- Multi-threaded SIMD processor
- Fermi Architecture
- SIMT ISA: PTX
- Memory Hierarchy
- Summary on GPGPU

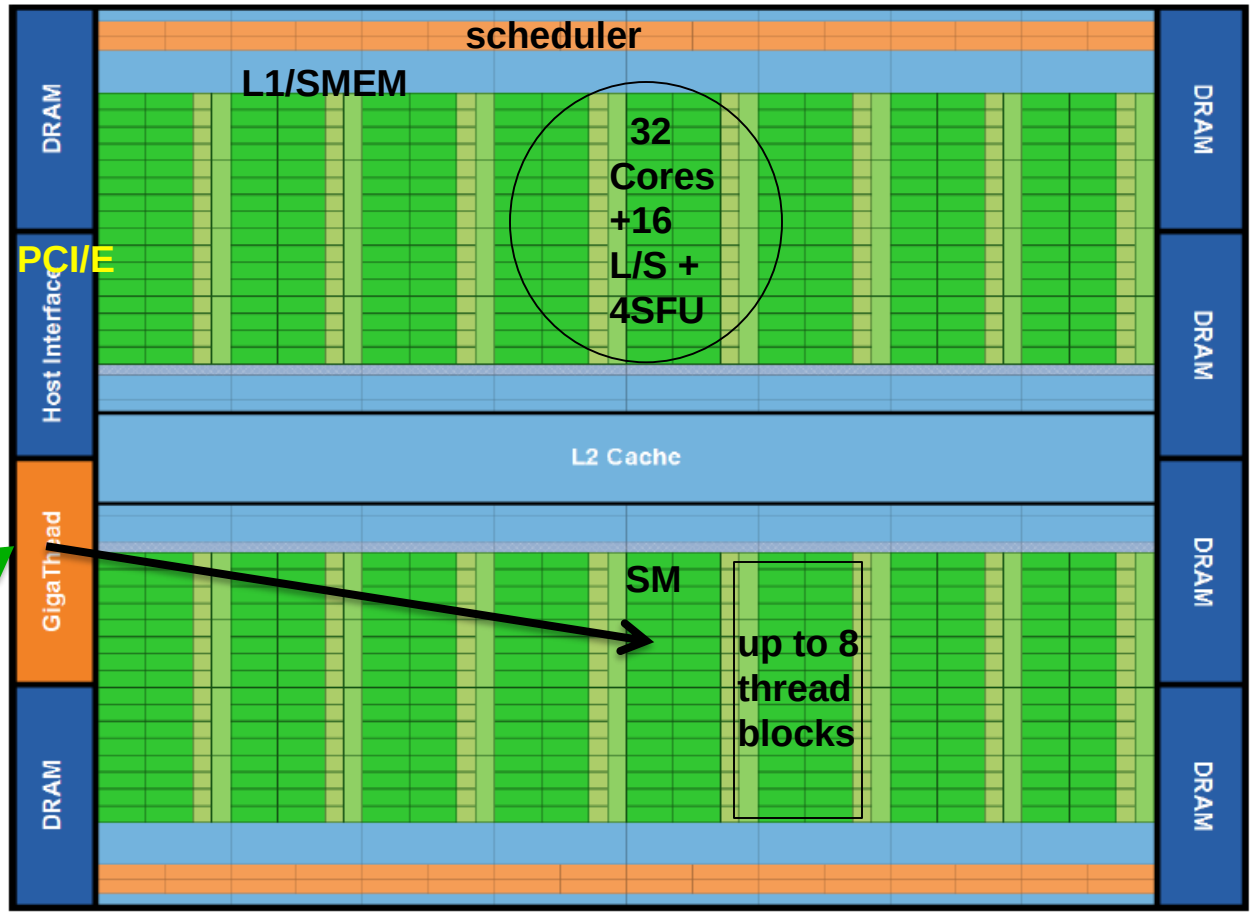
Fermi GPU Architecture- Floor plan

40-bit address space
40 nm TSMC
3 x 10^9 T
> Nehalem-Ex(2.3)
1.x GHz

16 multithreaded SIMD processors

Thread Block Distributor

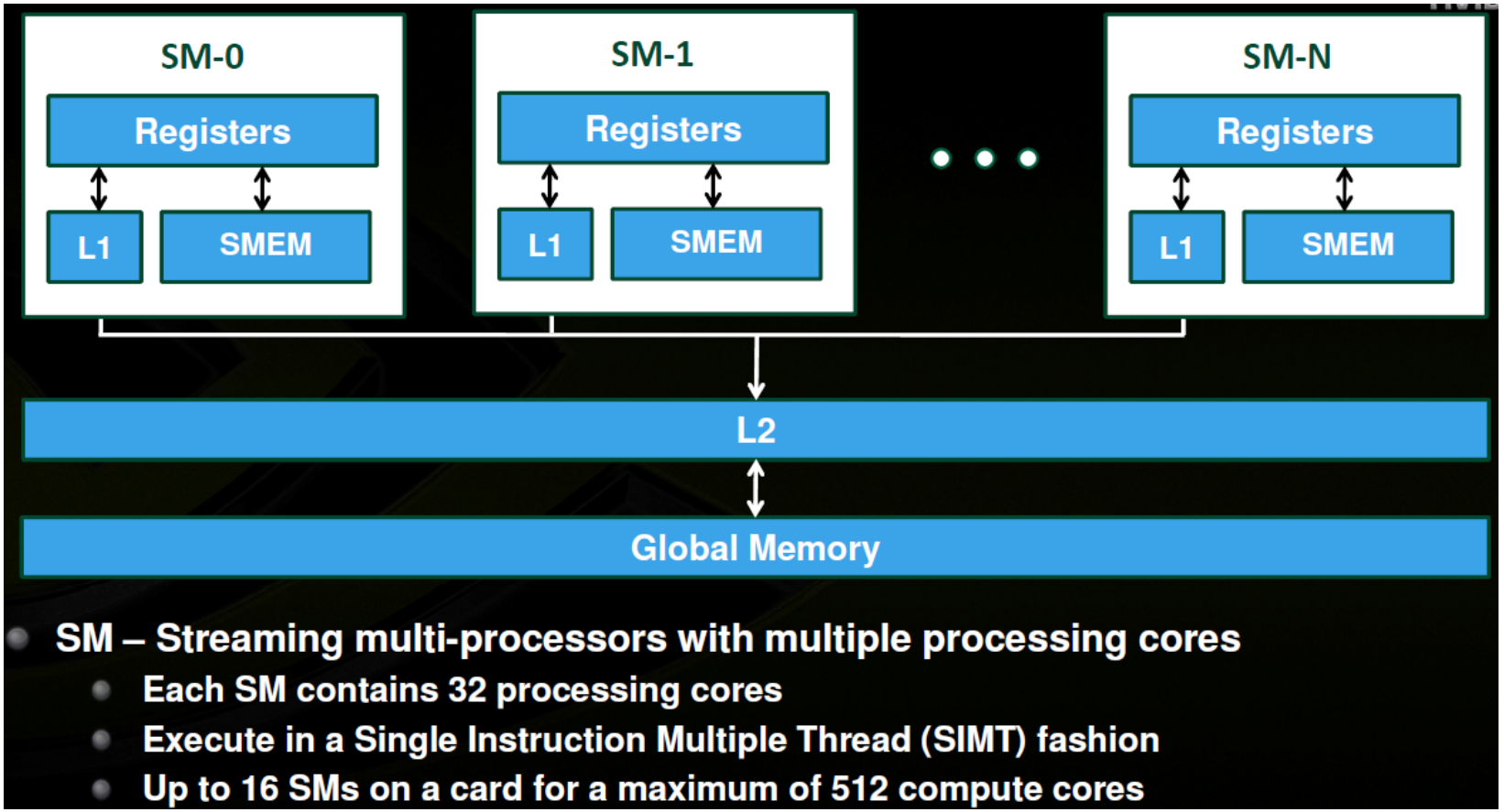
This GPU is a multiprocessor composed of multithreaded SIMD processors



Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

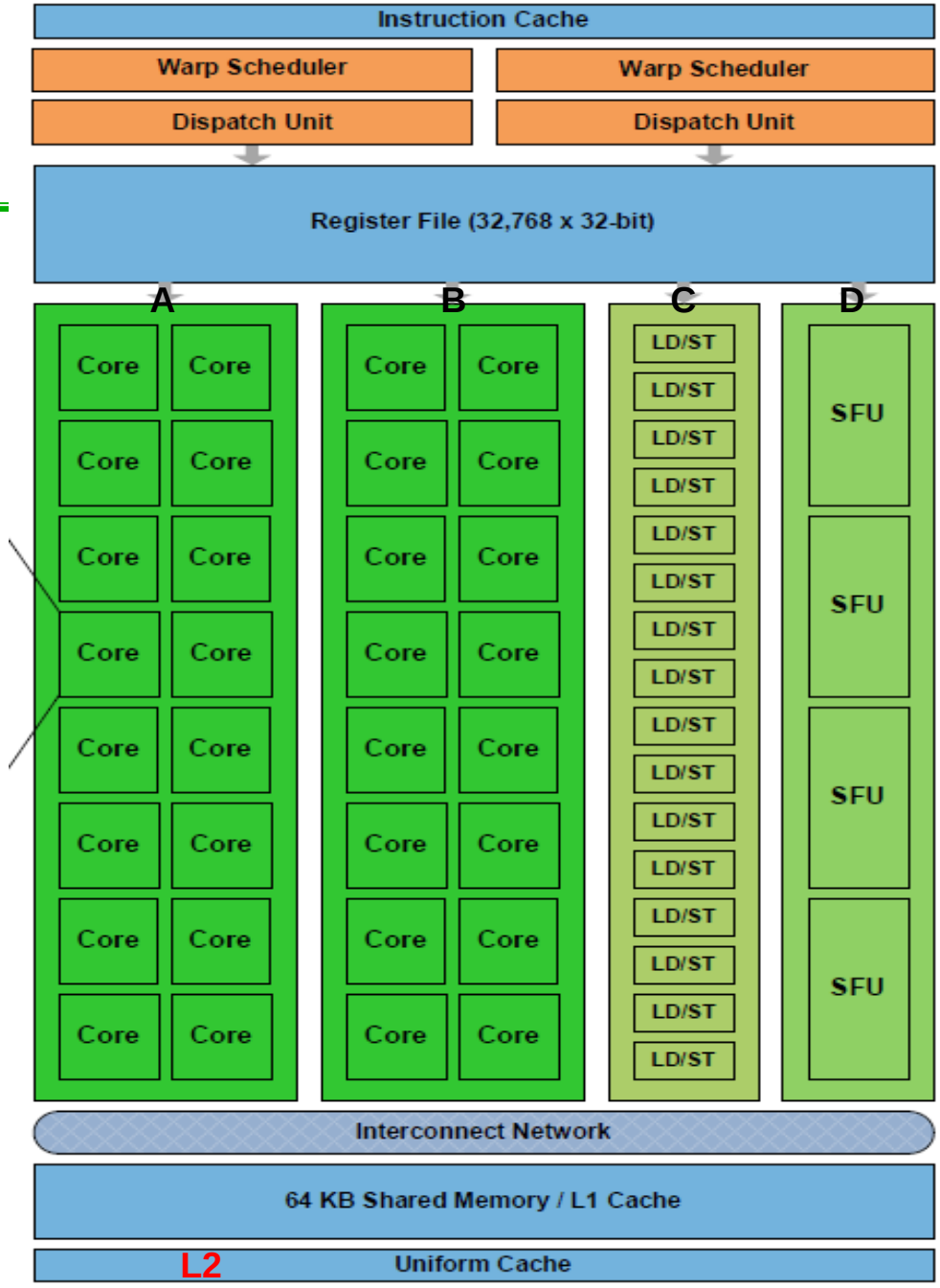
Fermi's Streaming Multi-processors

SMEM: shared memory in Fermi term, but this is actually a private local scratchpad memory for a thread block communication.
Data memory hierarchy: register, L1, L2, global memory
L1 + Local Scratchpad = 64KB configurable



A Steaming Multiprocessor ie., a Multithreaded SIMD Processor

- An SM consists of 32 CUDA cores + some 16 Load/Store unit + 4 special functional units
- Registers: 32K x words
- L1 data cache private to each SM
- L1 Instruction cache
- L2 unified for data and texture, instruction(?), shared globally, coherent for all SMs.
- Instruction dispatch
  - (A, B) fs
  - (A+B) fd
  - (A, C)
  - (B, C)
  - (A, D)
  - (B, D), (C, D), etc



Fermi Streaming Multiprocessor (SM)

Warp Scheduler in Fermi

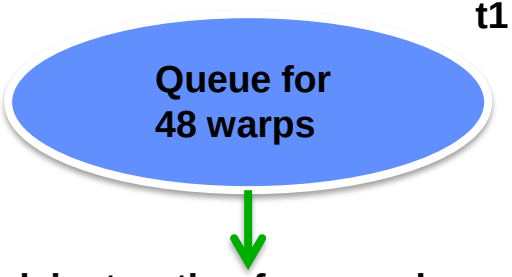
A warp is simply an instruction stream of SIMD instructions.

32 threads (on 32 SIMD lanes) have only one instruction stream due to SIMT!

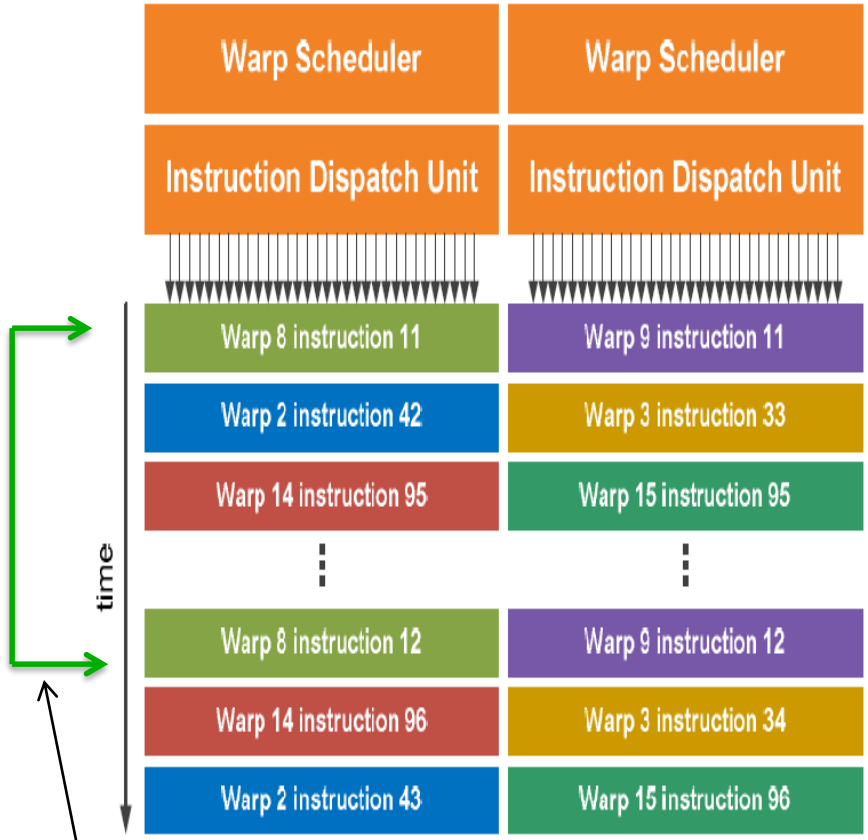
32 threads form a warp
Instructions are issued per warp
If an operand is not ready the warp will stall
- Context switch between warps when stalled
- Context switch must be very fast

- Fermi can have up to 48 active warps per SM => meaning 48 instruction streams.

48 x 32 = 1536 threads/SM



Pick instruction from ready warps



A cycle for issuing an instruction from warp 8

Outline

- Introduction
- Execution model
- Multi-threaded SIMD processor
- Fermi Architecture
- SIMT ISA: PTX
- Memory Hierarchy
- Summary on GPGPU

Mind-set behind NVIDIA Instruction Set Architecture : Parallel Thread Execution, PTX

- ISA is an abstraction of the hardware instruction set
- Hide the hardware instructions from the programmer
- Stable ISA across generations of GPUs
- PTX instructions describe operations on a single CUDA thread
- Usually map one-to-one with hardware instructions, but may have one-to-many and vice versa.
  - Translation to machine code is performed in software
  - Uses virtual registers

Software maps PTX to machine code

- **Translation to machine code is performed in software and at load time on a GPU**
  - **PTX uses virtual registers**
  - **A PTX instruction of a SIMD thread is broadcast to all SIMD lanes involved (hence a vector of n elements).**
  - **Compiler figures out how many physical vector registers a SIMD thread needs.**
  - **An optimizer divides the available register storage btw the SIMD threads**
  - **The assignment of virtual registers to physical registers occurs at load time (by MMU like mechanism?)**
  - **The optimizer also calculates places where branches might and places where diverged paths could converge.**

PTX Instructions

- **All instructions can be predicated by 1-bit predicate registers. RISC-type**
  - **Like ARM, conditional execution**
  - **setp.lt.f32. p, a, b // p = (a<b). Compare and set predicate**
  - **This is branch: @p bra target // if (p) goto target**
  - **barrier syc: bar.sync d // in a thread block, wait for threads**
  - **exit; terminate thread execution**
  - **call, ret**
  - **atomic read-modify-write**
  - **memory access, like ld/st**
  - **texture lookup**
  - **special functions; like square root, sine, cosine, log, exp….**
  - **arithmetic/logic,……**

Operation type

- **Untyped: 8, 16, 32, 64 bit; .b8,.b16, b32,.b64**
  - **(binary?, not one of the followings)**
- **Unsigned integer: .u8, .u16, .u32, .u64**
- **Signed integer: .s8, .s16, .s32, .s64**
- **Floating point: .f16,. f32, .f64**

Linpack benchmark: inner loop

- **X, Y are vectors of some length say 64; a is a scalar**

- **Y** = a x **X** + **Y**

Example: ALU and LD/Store

- the PTX instructions for one iteration of DAXPY

```
shl.u32      R8, blockIdx, 9    ; Thread Block ID * Block size (512 or 2^9)
add.u32      R8, R8, threadIdx          ; R8 = i = my CUDA thread ID
shl.u32, R8, R8, 3              ; R8 x 2^3
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])
```

Assign one CUDA thread to each loop iteration which gives unique thread block ID and thread ID in the given thread block.

Create 8192 CUDA threads of this copy (having different block ids) and uses the ID to address each element in the array

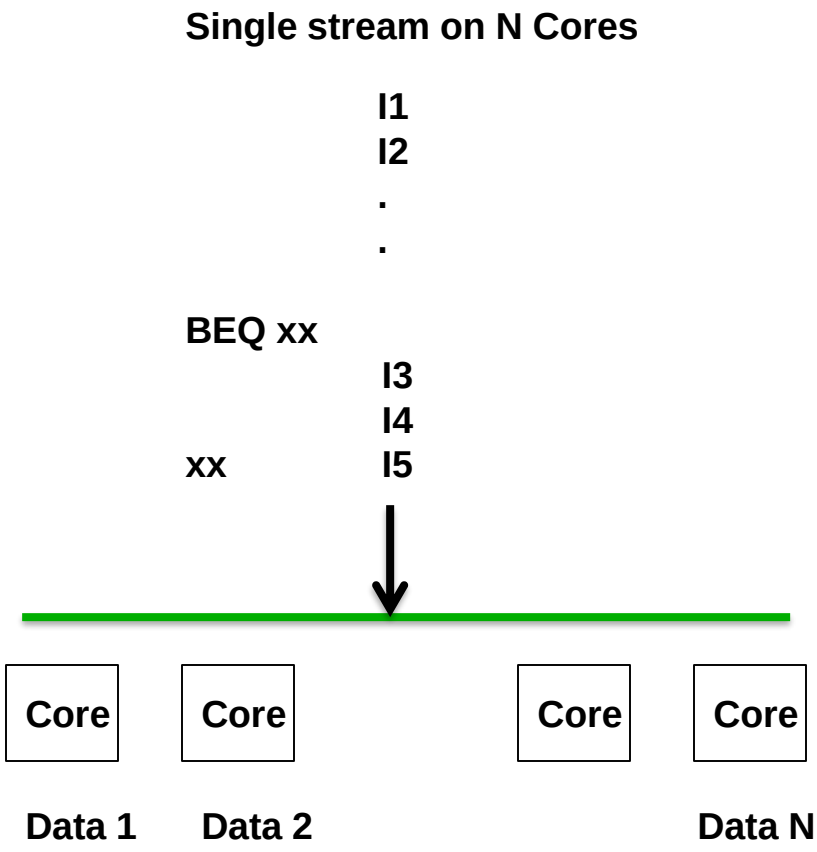
No need for incrementing or branching code

Address Coalescing Hardware

- **For data transfer from/to memory, a burst transfer of, say 32 sequential words is performed by the runtime hardware.**
- **To do this, the programmer must ensure that adjacent CUDA threads access nearby addresses at the same time so that they can be coalesced into one or a few memory blocks.**

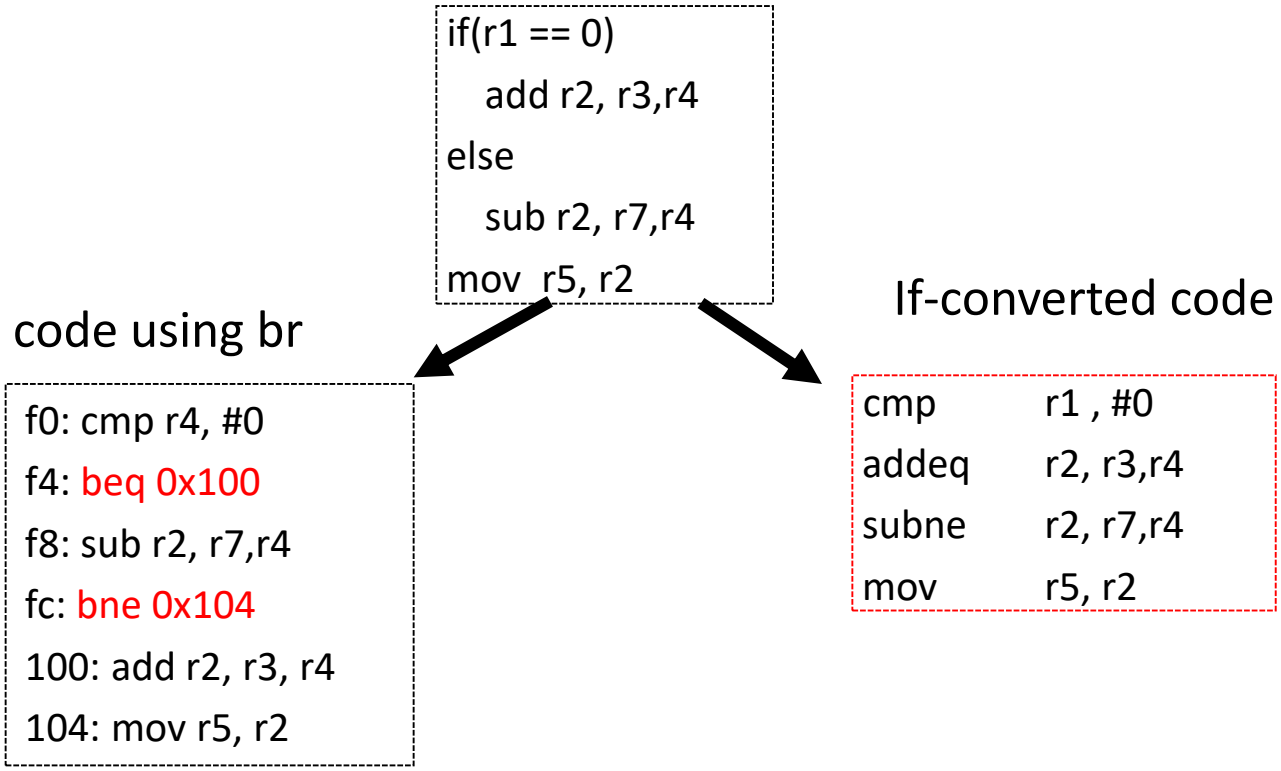
ISA issues for SIMT

- **Branch problem in SIMT**
  - **Can not use "regular branches" in SIMT because**
  - **If some gets I3 etc and some get I5,**
  - **then there is no single instruction stream anymore.**



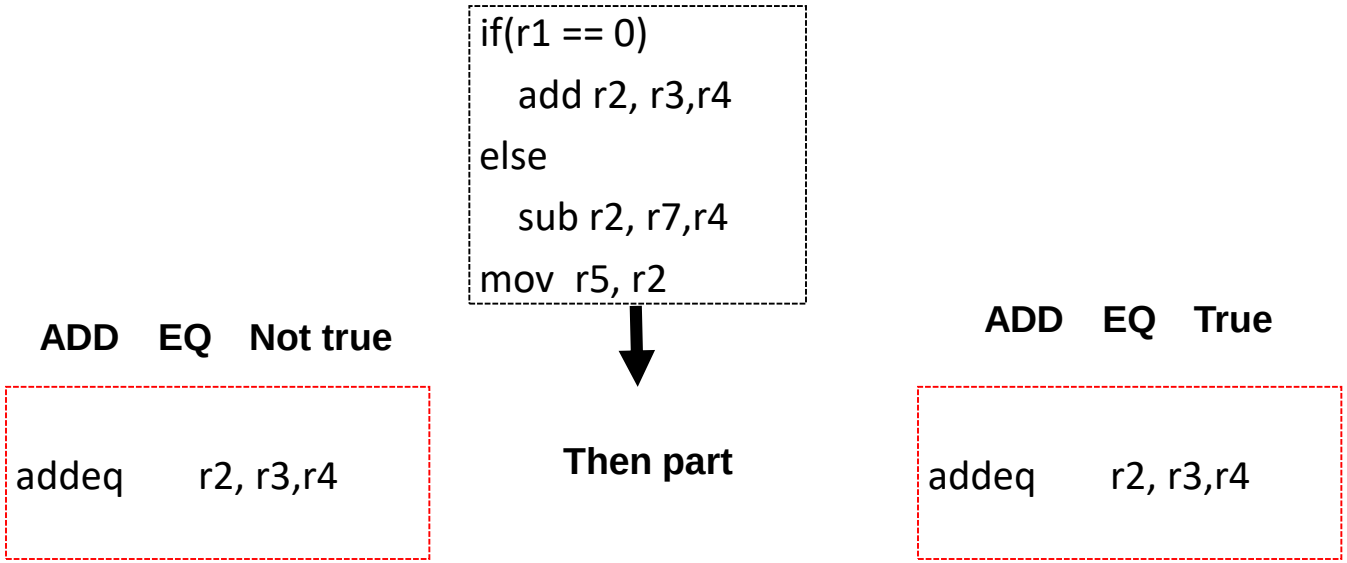
If-Conversion for Scalar

- If-conversion uses predicates to transform a conditional branch into a single control stream code.



If-Conversion for SIMT

- If-conversion uses predicates to transform a conditional branch into a single control stream code.



All lanes run the same instruction

Ways to deal with conditional execution-1

- **Predication if possible. PTX assembler optimizes an outer-level IF/THEN/ELSE coded with a PTX branch instruction to just predicated GPU instructions, without any GPU branch instruction.**

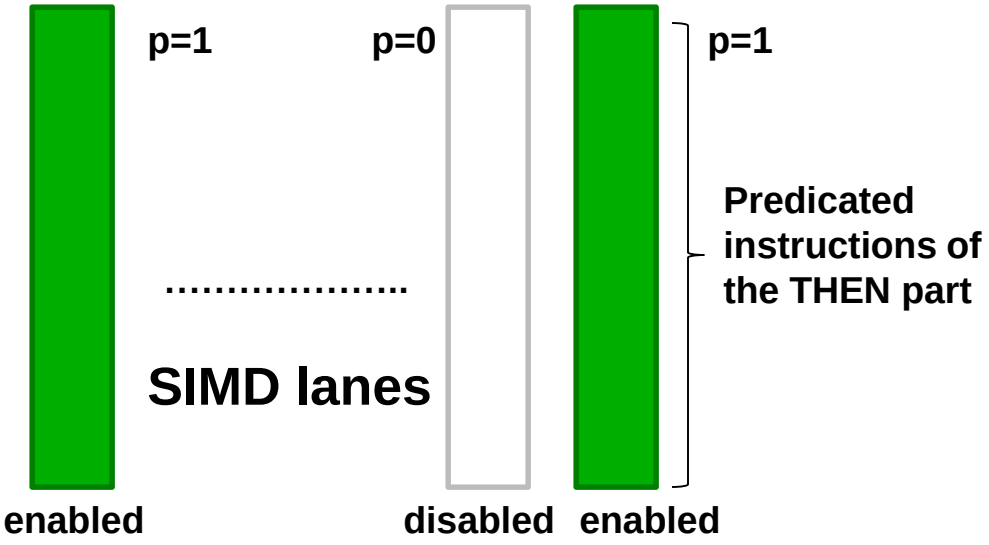
setp.lt.f32. p, a, b // p = (a<b)

p: per lane 1-bit predicate register

IF/THEN/ELSE
Step of execution
1. the THEN part is broadcast to all SIMD lanes (for p=1, green lanes store results)

2. Similarly, the ELSE part (for p'=1, white lanes store results)

Collectively, all ps' form a mask (1,1,1……0,1)



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); results stored depending on mask
  - 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

Ways to deal with conditional execution-2

- **Mixture of predication and GPU branch instructions with special stack instructions and markers that use branch synchronization stack to push a current active mask onto stack when a branch diverges.**
  - **Use branch instructions and branch synchronization markers to show the divergence and convergence of control flow;**
  - **however, all the same instructions are broadcast to all the SIMD lanes and conditionally executed based on their mask bits.**

branch synchronization marker: *push, *comp, * *pop executed by all SIMD lanes

```
if (X[i] != 0)
        X[i] = X[i] – Y[i];     // when p1 = 1
else X[i] = Z[i];               // when p1 = 0

ld.global.f64    RD0, [X+R8]          ; RD0 = X[i]
setp.neq.s32     P1, RD0, #0          ; P1= 1 if x[i] != 0
@!P1, bra        ELSE1, *Push         ; Push old mask, set new mask bits
                                      ; if P1 false, go to ELSE1 but actually nop
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
```

All SIMD lanes execute the same instructions in the IF-THEN-ELSE statement

Those SIMD lanes having p = 1, work and write result; others nop.

Those SIMD lanes having p = 0, work and write result; others nop.

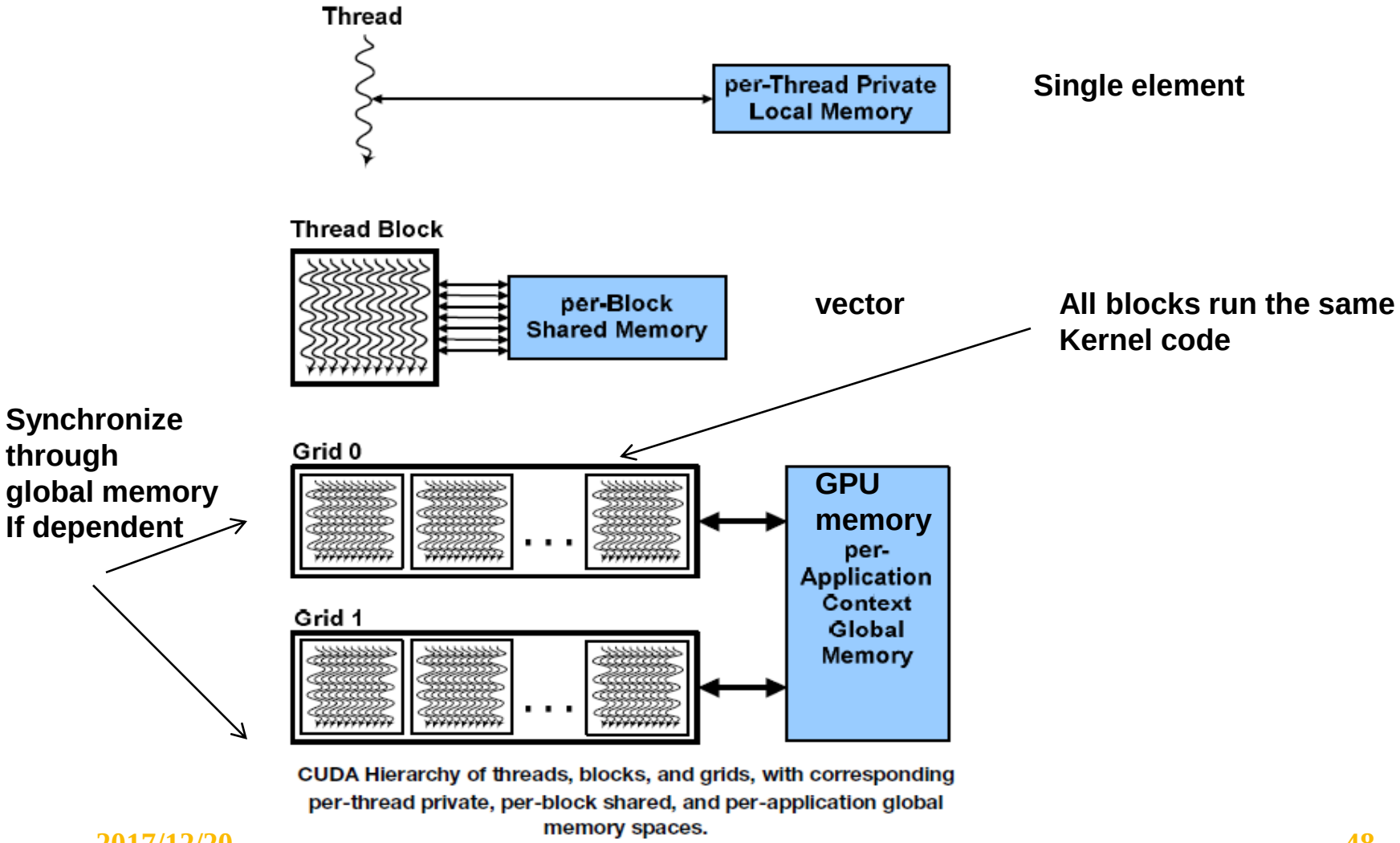
Illusion of MIMD branch-based program behavior on SIMD instructions

- **Illusion of some threads go one way, the rest go another.**
- **Illusion of a CUDA thread works independently on one element in a thread of SIMD instructions.**
- **In fact, each CUDA thread (each SIMD lane) is executing the same instruction either "committing their results" or "idle, i.e. no operation."**

Outline

- Introduction
- Execution model
- Multi-threaded SIMD processor
- Fermi Architecture
- SIMT ISA: PTX
- Memory Hierarchy
- Summary on GPGPU

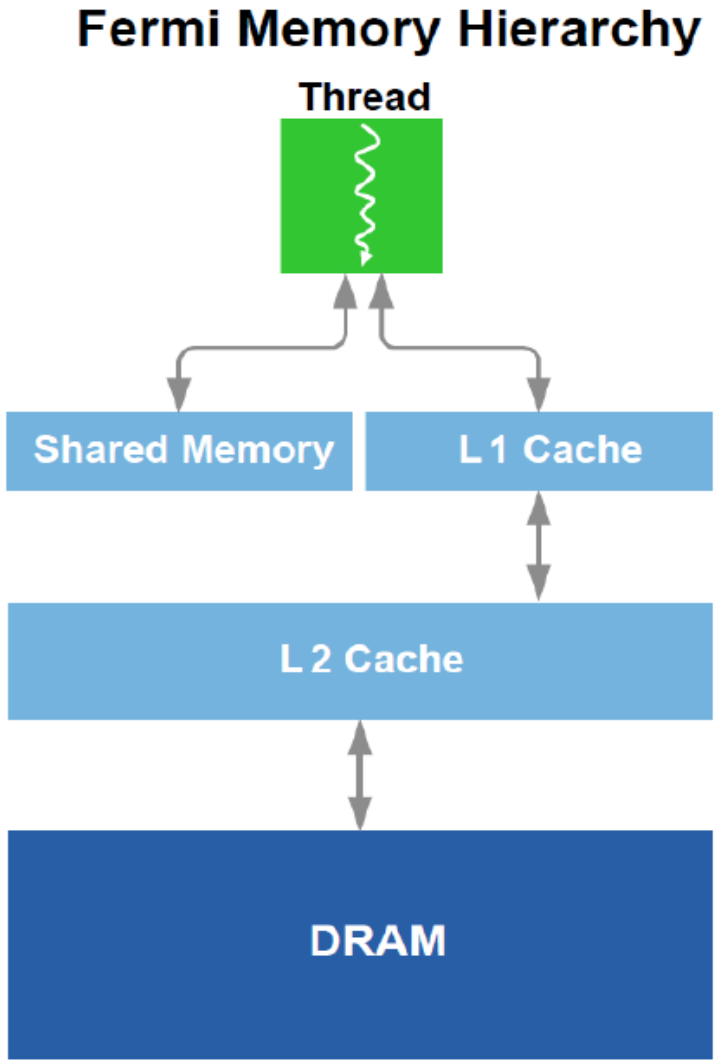
CUDA thread hierarchy



CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

Memory Hierarchy

- Similar to general purpose CPU
- Add a scratch-pad mem for group of threads that can locally share through load/store in the instruction stream-- a common DSP technique



NVIDIA GPU Memory Structures

- Each SIMD Lane (a CUDA thread) has private section of off-chip DRAM
  - "Private memory"
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory
  - Host can read and write GPU memory

Outline

- Introduction
- Execution model
- Multi-threaded SIMD processor
- Fermi Architecture
- SIMT ISA: PTX
- Memory Hierarchy
- Summary on GPGPU

NVIDIA GPU Architecture

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files
- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

Fermi Architecture Innovations

- Each SIMD processor has
  - Two SIMD thread schedulers, two instruction dispatch units
  - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Thus, two threads of SIMD instructions are scheduled every two clock cycles
- Fast double precision
- Caches for GPU memory
- 64-bit addressing and unified address space
- Error correcting codes
- Faster context switching
- Faster atomic instructions

What new in Fermi - April 2010

- **D. Patternson said in 2009**

	Top 10 Innovations in Fermi	*Top 3 Next Challenges*
1	Real Floating Point in Quality and Performance	The Relatively Small Size of GPU Memory
2	Error Correcting Codes on Main Memory and Caches	Inability to do I/O directly to GPU Memory
3	64-bit Virtual Address Space	No Glueless Multisocket Hardware and Software
4	Caches	
5	Fast Context Switching	
6	Unified Address Space	
7	Debugging Support	
8	Faster Atomic Instructions to Support Task-Based Parallel Programming	
9	A Brand New Instruction Set	
10	Also, Fermi is Faster than G80	

Unified Address Space

- **Patternson's word**

Past GPUs had a variety of different types of memories, each in their own address space. Although these could be used to achieve excellent performance, such architectures are problematic with programming languages that rely on pointers to any piece of data in memory, such as C, C++, and CUDA

Fermi has rectified that problem with by placing those separate memories—the local scratchpad memory, the graphics memory, and system memory—into a single 64-bit address space, thereby making it much more easier to compile and run C and C++ programs on Fermi.[6] Once again, PTX enabled this relatively dramatic architecture change without the legacy binary compatibility problems of mainstream computing.

Like caches, a unified address space increases the types of programs that can run well on GPUs.

Debug using GPU's thread!

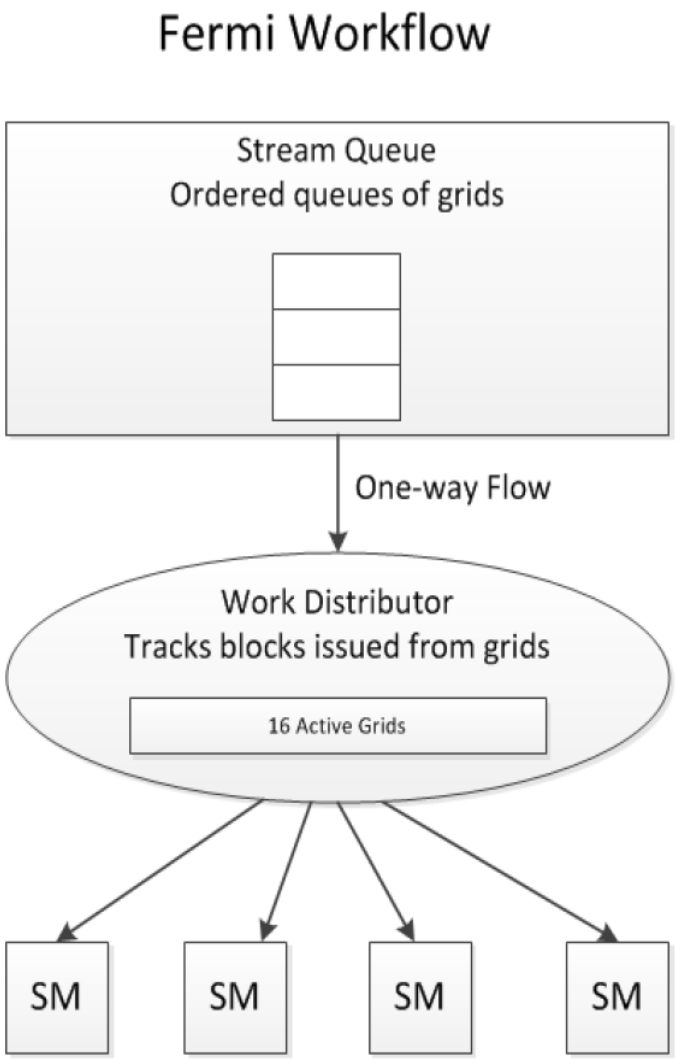
- **Patternson's word**

A significant change in Fermi is that traps, breakpoints, and so on are now handled in GPU trap handler software by GPU threads. GPU trap handler software can interact with CPU code as needed. That lets the GPU be more flexible, enables robust debugger actions, and lets the GPU provide system call support.

This innovation will likely popularize debuggers like GDB or Integrated Design Environments like Microsoft's Visual Studio for programming GPUs.

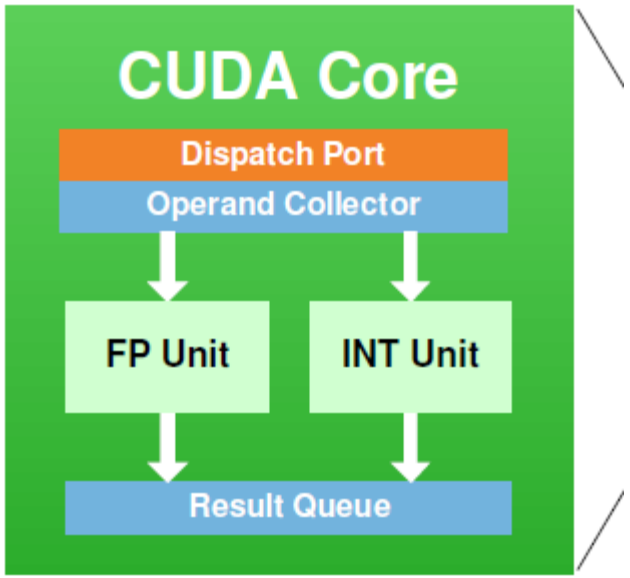
Work Model in Fermi

- **Host CPU launches a grid of thread blocks and let them run to completion.**
- **Unidirectional command flow.**



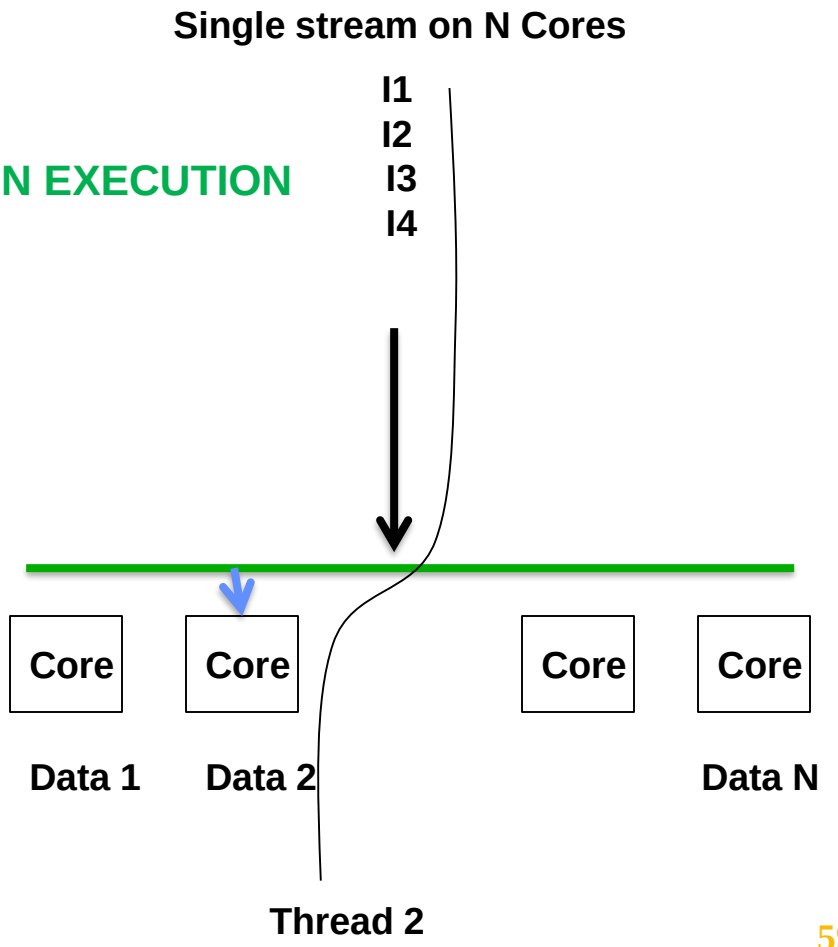
A CUDA Core

- **A core is a functional unit of some operations**
  - **Pipelined**
  - **Int unit: Boolean, shift, move, compare, convert, bit-field extraction, bit-reverse, etc.**
  - **FP unit: IEEE 754, fused MA etc.**
  - **How to effectively schedule instruction to a pipeline ?**



SIMT: Single Instruction Multiple Thread

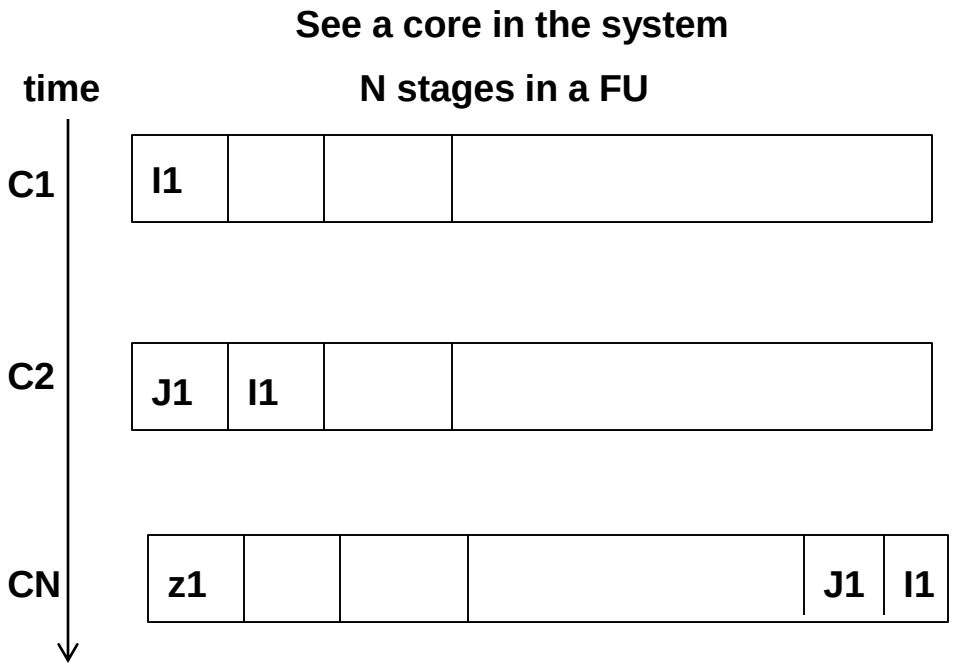
- Clarify what is what
- What is S?
- What are threads?
  - AN INSTRUCTION STREAM IN EXECUTION



Instruction Stream Scheduling and Pipeline

- An SM executes one or more thread blocks
- A group of N-threads called a warp. (Recall warp speed in Star Trek?)
- A warp scheduler issues (broadcasts) one instruction to either X cores (thus SIMD) or X Load/Store Units, or to Z SFUs.
- However, this is a pipeline functional unit!
- Assuming independent N instruction streams
- So guess: Longest pipeline length = 32?

- Actually a warp scheduler picks from 48 streams for instruction dispatching!



S1: I1 I2 I3 I4 I5…
S2: J1 J2 J3 J4 J5..

SN: z1 z2 z3….

After N cycles,
I1 completes
Warp back to
Issue I2 of S1, and etc.
So, if I2 depends on I1,
It has a room of N cycles
for execution latency.

Inside warp scheduler

- **Scheduling optimization: ILP & Hyper threading**
  - **Limited version of OOO**
  - **Register scoreboard: Allow OOO but stall on WAW and WAR hazards. Per stream view!**
  - **For RAW hazard, similar toTomasulo's basic. Per stream view.**
  - **Many instruction streams to dispatch through multiple warp schedulers. Simultaneous Multi-Threading !**

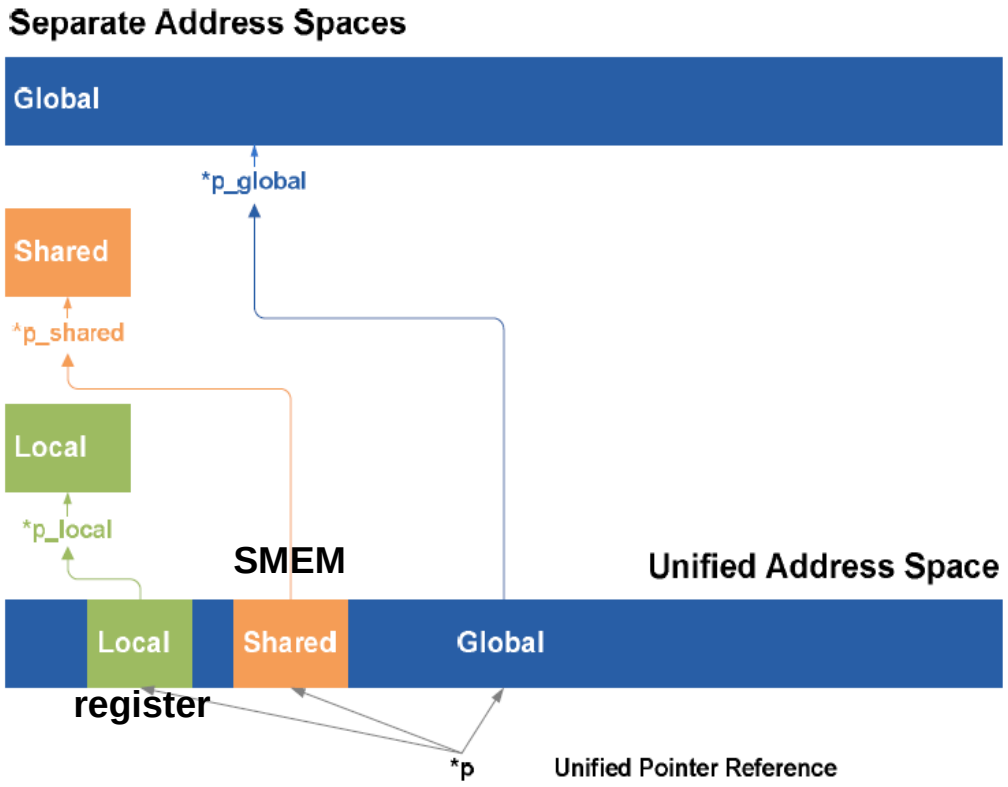
a) Register scoreboarding for long latency operations (texture and load)
b) Inter-warp scheduling decisions (e.g., pick the best warp to go next among eligible candidates)
c) Thread block level scheduling (e.g., the GigaThread engine)

However, Fermi's scheduler also contains a complex hardware stage to prevent data hazards in the math datapath itself. A multi-port register scoreboard keeps track of any registers that are not yet ready with valid data, and a dependency checker block analyzes register usage across a multitude of fully decoded warp instructions against the scoreboard, to determine which are eligible to issue.

Unified Address Space in Program View

- **A load/store directly accesses any type of the memory.**
- **A hardware translation unit maps load/store address to the correct memory location.**

With PTX 2.0, a unified address space unifies all three address spaces into a single, continuous address space. A single set of unified load/store instructions operate on this address space, augmenting the three separate sets of load/store instructions for local, shared, and global memory. The 40-bit unified address space supports a Terabyte of addressable memory, and the load/store ISA supports 64-bit addressing for future growth.



Unified address memory access by:

- **Hardware assisted page mapping that determines**
  - **which regions of virtual memory get mapped into a thread's private memory**
  - **which are shared across a block of threads**
  - **which are shared globally**
  - **which are mapped onto DRAM**
  - **which are mapped onto system memory**
- **As each thread executes, Fermi automatically maps its memory references and routes them to the correct physical memory segment.**

Resource Allocation in an SM

Registers and shared memory are allocated for a block as long as that block is active

- Once a block is active it will stay active until all threads in that block have completed
- Context switching is very fast because registers and shared memory do not need to be saved and restored

- **How many active threads to run depends on**
  - **How many registers to use for a thread**
    - **» since total has 32K registers**
  - How much SMEM to use for a thread

As usual, Compiler determines these allocations!

Resource Utilization in an SM

- **Utilization determined by:**
  - **How many registers are allocated to each active thread or to each instruction stream? (compiler)**
  - **How many SMEM are allocated to each thread? (compiler)**
  - **Each SM support s 8 active blocks and how big is the block size of each of the active blocks? Cannot be too small! (programmer??)**

✓ **Example**

a thread uses 21 registers, 32K/21 = 1560 threads

1560 > 1536 threads (spec)

Good utilization depends on the above 3 settings!

Need to see: FU utilization, throughput achieved, and bandwidth used

And in Conclusion

- ISA Architecture for GPU
  - ISA design, branch, predication, indexed Jump, etc
- SIMT Architecture
  - Multi-threaded SIMD processor
  - Whole GPU
  - Memory support
- Software
  - Compiler
  - PTX assembler and optimizer
  - Run time

Reference

- **The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges, *David Patterson Sept. 30, 2009.***

Whitepaper

NVIDIA's Next Generation
CUDA™ Compute Architecture:

Fermi™

Whitepaper

NVIDIA's Next Generation
CUDA™ Compute Architecture:

Kepler™ GK110